Regular Paper

The TABLET Programming Learning Environment: from Block-based to Text-based Programming

Takumi Miyajima[1,†1,a)] Hideya Iwasaki[1,†2,b)] Yasushi Kuno[1,c)]

Received: February 10, 2022, Accepted: May 18, 2022

Abstract: From 2020, programming is a compulsory subject in elementary schools in Japan. Since many schools are using block-based programming languages and environments as teaching materials, the number of students who have already learned block-based programming is expected to increase. To help such learners of block-based programming shift to text-based programming languages, we have designed and implemented a programming learning environment named TABLET. We have designed TABLET as a syntax-directed system that focuses on making the learner aware of the syntax of the target text-based language. To this end, TABLET incorporates two programming behaviors: deriving blocks for non-terminal symbols of the target language and writing program code directly with text. TABLET synchronizes these two behaviors to make it easier for the learner to grasp the correspondence between block-based programs and text-based programs. TABLET can be used for many target text-based languages as long as the syntax can be given as a set of Backus-Naur Form (BNF) rules. Thus, TABLET is general enough to capably generating blocks for the language and to enable the both programming behaviors. We conducted evaluation experiments with second-year undergraduate students. We found that TABLET made it easier for the students to grasp the correspondence between the block-based and text-basd programs and to understand the syntax of the target language.

Keywords: Text-Based Languages, Block-Based Languages, Syntax-Directed Systems, Structural Editors, Programming Environments

1. Introduction

Programming languages can be classified into two main types: *text-based languages* and *visual languages*. A program in a text-based language is created as a sequence of characters that obeys its syntax. Most programming languages, such as C and Java, are text-based languages. In contrast, visual languages use visual components such as blocks and pictures, and combine them with screen operations to create programs. Scratch [1], [2], [3], Blockly [4], [5], Viscuit [6], [7], [8] are examples of visual languages. Among visual languages, this paper focuses on those that use blocks. Hereafter, in addition to such languages, languages in which programs are created by structural editors that use blocks so as to make the programs follow the syntactic structures are referred to as *block-based languages*.

From 2020, programming is a compulsory subject in elementary schools in Japan [*1]. In this context, block-based languages, which are easy to understand visually and intuitively, are considered to be suitable for elementary school students, and many elementary schools are introducing them. In fact, the Ministry of Education, Culture, Sports, Science and Technology in Japan uses block-based language in practical examples such as "drawing regular polygons" and "tools using the properties and functions of electricity" [*2,*3]. It is thus expected that the number of students who have already learned block-based languages will increase in the near future.

However, programming languages commonly used in practical program development today are text-based languages, which are more descriptive than block-based languages because program code is written using keyboard inputs rather than mouse operations. Thus, to perform practical programming in the future, programming beginners, who have learned the basic programming in block-based languages, are required to *shift* to text-based languages equipped with superior descriptive power. Here, to "shift" means to become able to make programs in a text-based language by inputting text from the keyboard and fully understand and be aware of the *syntax* of the language. However, there is high burden to shift because large differences exist between block-based and text-basd languages. We hypothesize that this burden is due in large part to the change from programming that does not require awareness of language syntax to programming that does.

To solve the aforementioned problems and to support a smooth shift from block-based languages to text-based languages, we design and implement TABLET (Text And Block programming Learning EnvironmenT). TABLET is intended for the learner who

[1] Graduate School of Informatics and Engineering, The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[†1] Presently with PCI Solutions INC.
[†2] Presently with School of Science and Technology, Meiji University
[a)] kumikumi.core5@gmail.com
[b)] hideya.iwasaki@acm.org
[c)] y-kuno@uec.ac.jp

[*1] https://www.mext.go.jp/content/20200218-mxt_jogai02-100003171_002.pdf (in Japanese) (accessed 2022-05-31).
[*2] https://www.mext.go.jp/component/a_menu/education/micro_detail/__icsFiles/afieldfile/2019/05/21/1416331_001.pdf (in Japanese) (accessed 2022-05-31).
[*3] https://miraino-manabi.mext.go.jp/ (in Japanese) (accessed 2022-05-31).

has already learned some block-based language but is in the early stage of learning a text-based language. The complexity of the syntax of a text-based language varies from language to language; we assume simple languages.

TABLET aims to lower the hurdles due to the following two changes that occur with the shift:
- a change from programming without awareness of the grammar of the language to programming with awareness of the grammar, and
- a change from mouse operations to text input operations.

To this end, we set the following three goals of TABLET:
- making the learner be aware of the syntax of the target text-based language,
- making it easier to understand the correspondences between blocks and texts, and
- reducing the hurdle of keyboard input.

To achieve these goals, TABLET uses the following two synchronous basic operations for constructing a program in the target text-based language: block-combining operations using the mouse and text-inputting operations using the keyboard. This makes it easy to understand the correspondences between blocks and texts, aiming at both the integration of the two and to ease the shift.

Furthermore, TABLET is able to generate blocks on the basis of a given BNF of the text-based language to be shifted. This feature makes it easier for the learner to be aware of the syntax of the target language, and at the same time, makes TABLET general enough to cope with a variety of text-based languages.

This paper describes the design of TABLET, an overview of learning programming with TABLET, the implementation of TABLET, and the evaluation of TABLET through experiments with testee students.

This paper is organized as follows. Section 2 describes the design of TABLET. Section 3 gives an overview of TABLET with concrete examples of its use. Section 4 describes the implementation of TABLET. Section 5 describes experiments to evaluate TABLET and obtained results. Section 6 describes related work, and Section 7 concludes the paper.

2. Design of TABLET

This section describes the details of the TABLET design.

In designing TABLET, we considered that the major barrier that hinders the learner from shifting from a block-based language to a text-based languages lies in the difficulty in the awareness of the syntax structures of the text-based language. Therefore, we designed TABLET as a type of structural editor, which makes the learner aware of the syntax by enabling to create only programs that follow the syntax. In addition, we designed TABLET to immediately synchronize the block code and text code to support the learner's understanding of the syntax by making it easier to grasp the correspondence between both types of code.

2.1 Syntax-Directed System

To overcome the barriers previously described, we designed TABLET as a system that makes the learner aware of the syntax of a target text-based language; it is a type of a syntax-directed structural editor that functions on the basis of the syntax rules of the language. Given the BNF of the language, TABLET generates blocks for non-terminal symbols to make it possible to perform derivations by block operations on the basis of the syntax rules.

In block-based languages such as Scratch, where notches and bumps exist in the blocks, the learner creates a program so that the unevennesses of blocks match. We were concerned that designing the blocks of TABLET in such a way would diminish the need for the learner to be aware of the syntax of the text-based language.

Generating blocks from the BNF enables the code to be created through derivation operations of blocks. Because derivation operations need to follow syntax rules, the learner ultimately becomes aware of the syntax. Furthermore, for any given block, it is possible to provide the text to replace from the keyboard. This enables the range of the syntax to be adjusted appropriately in accordance with each learner's achievement level of the shift to the target text-based language.

2.2 Integration of Blocks and Texts

Another design that TABLET uses to aim at removing the learner's barrier is to integrate both blocks and texts. To this end, TABLET provides both block operations and text operations, enabling the learner to program using either blocks or texts, or both. Furthermore, the results of operations on one side are immediately reflected on the other side's code, making the system bidirectional. By combining blocks to build the entire structure of the program and giving its details in texts, the learner can grasp the correspondence between blocks and texts, and can also become used to text-based programming. As an auxiliary feature, TABLET is able to display the syntax tree of a program in progress (or already created) to help the learner visually learn the syntax.

Forcing the learner to write a program directly by using only texts from the beginning is a high barrier because in addition to the transition from the mouse to the keyboard, it requires the comprehension of the text-based language's syntax. Therefore, by using both block operations based on the mouse and text operations based on the keyboard, we have achieved a system that learners are familiar with and made it possible to narrow the range of the syntax the learner pays attention to.

It was also considered to make TABLET offer only block operations like Blockly. However, even though the system displays the corresponding program text in a text-based language to review, the learner would have no programming opportunity in the text-based language. We believed that such a system would be less effective for supporting the shift to text-based languages.

Thus, to make TABLET easy to use for the learner of block-based languages and to let the learner be accustomed to text-based languages, we believed that an integration of both blocks and texts would be suitable. Since mouse and keyboard operations can be performed simultaneously, it is expected that the learner will have more opportunities to use the keyboard and, as a result, the hurdle of the text input will become lower. If the notation of the block code and that of the text code are far apart, it is difficult for the learner to grasp the correspondence between them. Thus, the blocks are generated from the syntax of the text-based language. This point will be discussed in Sections 3.2 and 3.3.

2.3 Generality of TABLET

TABLET focuses on the editor function for creating a program, not running it. For this reason, TABLET does not adopt processing, e.g., compiling a program, tailored to some specific text-based language nor does it provide an integrated development environment from program creation to execution.

We limited TABLET's language-specific functions only to creating blocks and generating a parser from the BNF of the target language. As will be described in Section 4.1, the BNF is provided in the form of SableCC's syntactic descriptions [9], [10]. Although the syntax of the target language is thus limited to LALR(1) used by SableCC, TABLET is not a language-specific environment; it is general enough to be used for a variety of languages within the aforementioned range as long as the BNF is prepared. This generality makes it possible, for example, to learn a procedural language step by step, increasing the number of control structures, or to use multiple text-based languages in accordance with the learning objectives and/or instructor's policies.

There have been many studies on systems that are capable of automatically generating a structural editor from a given language's syntax, including the system by Arefi et al. [11], ASF+SDF Meta-environment [12], LISA [13], MPS [14], and the system by Ferreira [15]. However, to the best of our knowledge, there is no system like TABLET that supports a variety of LALR(1) languages, enables text input of arbitrary non-terminal symbols, and synchronizes graphical and textual representations of the program.

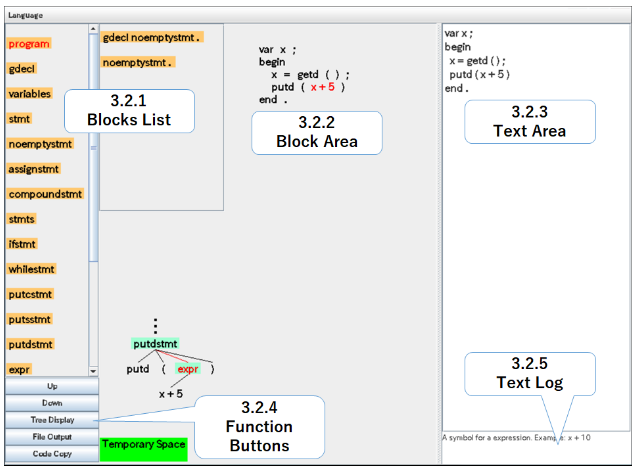
3. Overview of TABLET

This section describes the basic operations of TABLET. We then give an overview of TABLET by using two concrete examples of simple text-based languages: a language based on four arithmetic operations (hereafter, a four arithmetic language) and a Pascal-like procedural language. As described in Section 2, TABLET is not an integrated development environment that is capable of handling everything from program creation to its execution. TABLET is a system that uses blocks and texts together and outputs the textual program of the target text-based language to an external file.

3.1 Programming Style in TABLET

In TABLET, the learner creates a program by repeatedly replacing non-terminal symbols (including identifier name, constant literal, and other symbols usually treated as lexical tokens) with more concrete symbols on the basis of the BNF syntax rules for the target text-based language.

Blocks in TABLET are objects for non-terminal and terminal symbols displayed in the Blocks List (Section 3.2.1) and the Block Area (Section 3.2.2) on the screen. Only blocks for non-terminal symbols are targets of replacement. Blocks for non-terminal symbols other than those treated as tokens can be replaced by block operations with the mouse (Section 3.3.1) or by text operations in Text Area (Section 3.2.3) with the keyboard (Section 3.3.2) that directly enters text matching the syntax of



```
var x ;
begin
  x = getd ( ) ;
  putd ( x + 5 )
end .
```

```
var x ;
begin
 x = getd ( ) ;
 putd ( x + 5 )
end .
```

A symbol for a expression. Example: x + 10

Fig. 1 Screen of TABLET.

the non-terminal symbol. Replacement by a block operation is equivalent to performing a *derivation* from the left-hand side to the right-hand side of the syntax rule with respect to the non-terminal symbol to be replaced. In addition, non-terminal symbols for tokens can only be replaced by providing text from the keyboard. TABLET displays blocks in colors so that the learner can grasp at a glance what operations are possible for them. An explanation is given in Section 3.3.3. In the examples shown in Sections 3.4 and 3.5, each block is enclosed in a rectangle frame and a symbol sequence in Text Area is bracketed by 《 》.

Blocks are named after the BNF description of the syntax rules. In contrast, many existing block-based languages represent blocks in a natural language; a learner familiar with existing block-based languages may find it difficult to read TABLET's block code. TABLET compensates for this shortcoming by displaying a natural language explanation of the target block in Text Log (Section 3.2.5) or pop-up window, as described in Section 3.3.4.

3.2 Screen of TABLET

Figure 1 shows the screen of TABLET. We describe each component of the screen.

3.2.1 Blocks List

The large area on the left shows Blocks List, a list of blocks, each of which corresponds to a non-terminal symbol except tokens, in the syntax rules of the target text-based language. A block can be selected by clicking on it with the mouse; the selected block is indicated by red text. The small area on the right of Blocks List displays a list of block sequences, or Block Sequences List, each of which can replace the block selected from Blocks List. Each block sequence corresponds to each of the right-hand side of the syntax rule of the selected block.

3.2.2 Block Area

This is the area where a program is created by block operations. In this area, the mouse is used. A block operation performed in Block Area is immediately reflected in the program displayed in Text Area, which is described in the following.

3.2.3 Text Area

This is the area where a program is created as texts. In this area, the keyboard is used to create the program. The content of the program displayed in Text Area is essentially the same as that displayed in Block Area, except that the former only shows the contents of symbols that are not composed of blocks, whereas the latter consists of blocks. The results of code creation operations in Text Area are immediately reflected in Block Area.

3.2.4 Function Buttons

A block in Block Area can be selected by clicking it with the mouse.

Up Derived blocks containing the selected block are replaced with the pre-derived block, i.e., the block on the left-hand side of the syntax rule. If no pre-derived block exists, nothing happens.

Down The selected block is replaced with the derived blocks. If no derived blocks exist, nothing happens.

Tree Display The created program is displayed in the lower left corner of the block area in the form of a tree structure. Selecting it again hides the tree.

File Output The entire program is saved to the specified file.

Code Copy The entire program is copied to the clipboard.

3.2.5 Text Log

This is the area where the explanations of syntax elements and error messages signaled by the parser are displayed.

3.3 System Operations

In Block Area on the left-side of the TABLET screen, a program is created by block operations, and in Text Area on the right-side, it is created by text operations. Both contents are synchronized in real time, i.e., changes made in one area are immediately reflected in the other.

In Block Area, each block represents a symbol in the syntax rules of the target text-based language. For terminal symbols and non-terminal symbols for tokens that have already been entered, their contents appear as text directly in the blocks with a colorless background. When TABLET is started, the initial block code consists of the block for the start symbol.

3.3.1 Dragging and Dropping Blocks

The derivation operation from the left-hand side to the right-hand side of a syntax rule is performed by dragging and dropping blocks. Selecting a block in the Blocks List displays a Block Sequences List corresponding to the right-hand side of the syntax rule for the non-terminal symbol of the selected block. One of the block sequences can be selected and dragged with the mouse. Dropping it onto the block for the non-terminal symbol (in the block code) corresponding to the left-hand side of the syntax rule causes the block to be replaced with the block sequence. Such a *drag-and-drop* operation replaces a block in the block code. When a replacement is done, Text Area updates its contents at the same time.

3.3.2 Inputting Texts

Text Area displays the symbols that are the result of block operations. Editing by text input can be done by selecting one of the non-terminal symbols to be replaced in Text Area and typing its content from the keyboard. After entering the text, pressing the "Enter" key completes the replacement as long as the input is syntactically correct for the symbol. If the input contains a syntax error, a message is displayed in Text Log indicating that an error has occurred and no replacement is made. When edits are made in the text, they are reflected synchronously in Block Area.

3.3.3 Highlighting Symbols

Symbols that can be replaced by drag-and-drop operations or by text inputs are indicated by light blue blocks. However, symbols that cannot be further replaced by drag-and-drop operations and can be replaced only by text operations are shown in blue. Examples of such symbols are identifier names and constant literals, which are usually treated as tokens.

A target block of an operation, such as a block stacked with a dragged symbol sequence or a block to be replaced by keyboard input, is indicated by red text. While a replacement text is being entered from the keyboard, the incomplete text string is displayed in red on Text Area. Edited blocks including those for terminal symbols that cannot be further replaced are displayed with a colorless background.

3.3.4 Explanation of Symbol

Hovering the mouse cursor over a block on Block Area displays a pop-up with a description of the symbol for the block. In Text Area, when a symbol is selected by a mouse click, a description of the symbol and concrete examples are displayed in Text Log. These explanations help the learner understand the general meaning of a symbol.

3.3.5 Undo and Redo of Block Operations

After a replace operation by a drag-and-drop, the replacement can be undone and the original block can be restored by clicking on the replaced block (or one of the blocks if the replacement was made to a block sequence) with the mouse and then clicking on the "Up" function button. To restore the replacement again, the "Down" function button is clicked after selecting the block before replacement. "Up" and "Down" correspond to undo and redo, respectively.

3.3.6 Tree View of Blocks

By selecting the "Tree Display" function button, the block code can be displayed in the lower left corner of Block Area in the form of a tree structure. Selecting it again hides it. The highlighted symbol is placed at the center, and an upper and lower level are displayed; the other levels are omitted and displayed as "...". This function helps the learner visually grasp the structure of the code being created.

3.3.7 Temporary Saving of Code Fragments

If there is a code fragment in the already-created block code that the learner wants to temporarily save, the learner can select it with the mouse, drag it, and drop it on the "Temporary Space" in the lower left of Block Area. Multiple code fragments can be saved, and they are displayed in the block list by selecting the Temporary Space. By selecting and dragging one of the code fragments displayed in the block list, and then dropping it onto the block of a symbol that the derivation is applied to in the block code, the learner can replace the symbol with the stored code fragment.

3.3.8 Switching Display Language

By selecting the "Language" menu in the upper left corner, a list of natural languages supported by TABLET is displayed. By selecting one of them, the language displayed on the buttons, Text Log, and pop-ups switches to the selected language. TABLET currently supports Japanese and English, and the default is Japanese. All screen snapshots in the figures in this paper were taken when English was selected.

3.4 Example: Four Arithmetic Language

This section creates a program "`a = 2 * 3`" in a simple four arithmetic language. **Figure 2** shows its syntax in the BNF.

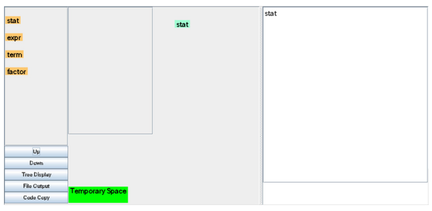
The startup screen of TABLET for this language is shown in **Fig. 3** (a). From here, the process for creating the program is shown from Fig. 3 (b) to Fig. 3 (k), and the completed program is shown in Fig. 3 (l).

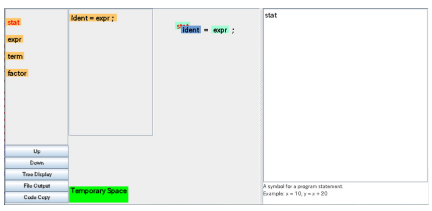
Blocks List is displayed on the left end of Block Area, and the block `stat` corresponding to the starting non-terminal symbol ⟨stat⟩ is displayed in the center as the initial block code. The symbol ⟪`stat`⟫ is also displayed in Text Area. Each block in Blocks List corresponds to a non-terminal symbol that is not treated as a token. If `stat` is selected from Blocks List, a Block Sequences List in which each block sequence corresponds to symbols derivable from ⟨stat⟩, is displayed. In this case, since only a single symbol sequence "⟨ident⟩ `=` ⟨expr⟩ `;`" exists on the right-hand side of the syntax rule of ⟨stat⟩, only this is displayed. By dragging `ident` `=` `expr` `;` using the mouse and dropping it on `stat` in the block code (Fig. 3 (b)), we can replace `stat` in the block code with `ident` `=` `expr` `;`. In Text Area, ⟪`stat`⟫ is also replaced with ⟪`ident = expr ;`⟫ at the same time (Fig. 3 (c)). If we perform similar operations on `expr` (Fig. 3 (d)), `term` (Fig. 3 (e)), and `factor` (Fig. 3 (f)), we can obtain the block code shown in Fig. 3 (g). In these operations, since every `expr`, `term`, and `factor` has multiple choices on the right-hand side of its syntax rule, multiple block sequences are displayed; we select an appropriate one among them, and perform the drag-and-drop operation.

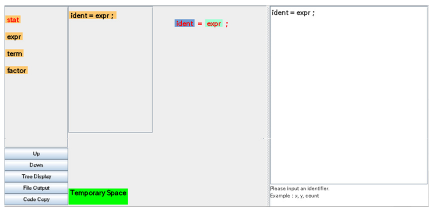
Next, we select ⟪`ident`⟫ on Text Area by clicking the mouse on it. Then, its corresponding block `ident` on Block Area turns to red (Fig. 3 (h)). We then enter "a" in Text Area and press the "Enter" key; both contents in the block and text code are updated synchronously to "a," and the block turns from blue to colorless (Fig. 3 (i)). We perform a similar text operation to symbol

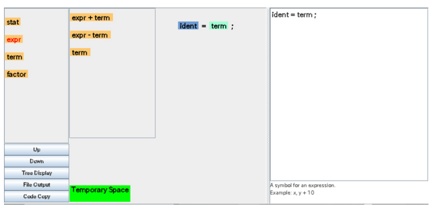
⟨stat⟩ ::= ⟨ident⟩ `=` ⟨expr⟩ `;`
⟨expr⟩ ::= ⟨expr⟩ `+` ⟨term⟩ | ⟨expr⟩ `-` ⟨term⟩ | ⟨term⟩
⟨term⟩ ::= ⟨term⟩ `*` ⟨factor⟩ | ⟨term⟩ `/` ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ ::= ⟨ident⟩ | ⟨iconst⟩ | `(` ⟨expr⟩ `)`
⟨ident⟩ ::= an alphanumeric string beginning with an alphabetic character
⟨iconst⟩ ::= a sequence of digits

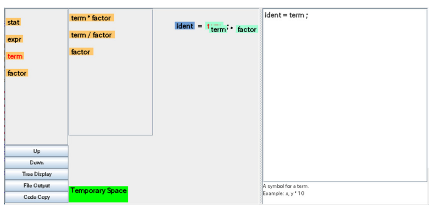
Fig. 2 Syntax of four arithmetic language.

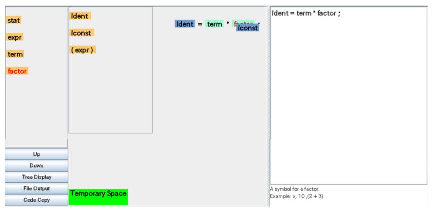

(a) Immediately after startup.

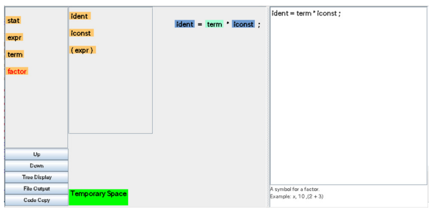

(b) Before replacing `stat`.

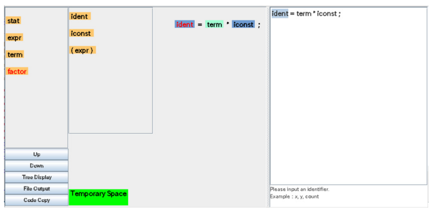

(c) After replacing `stat`.

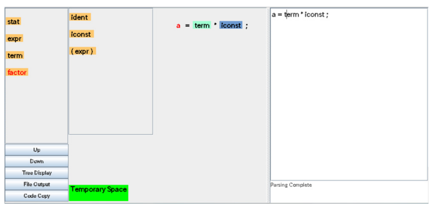

(d) Replacing `expr`.

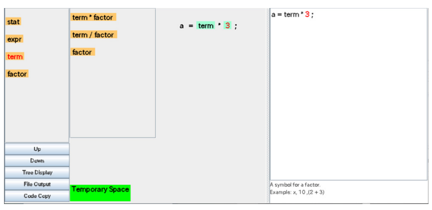

(e) Replacing `term`.

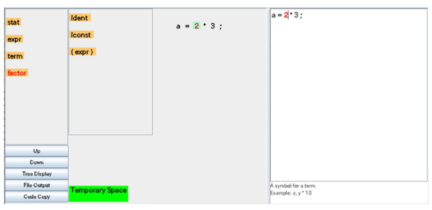

(f) Replacing `factor`.


(g) After completing block operations.


(h) Before entering ⟪`ident`⟫.


(i) After entering ⟪`ident`⟫.


(j) Entering ⟪`iconst`⟫.


(k) Entering ⟪`term`⟫.

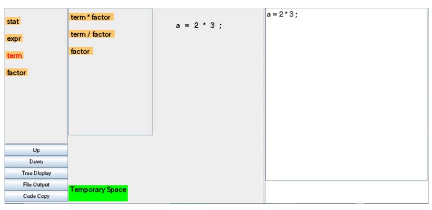
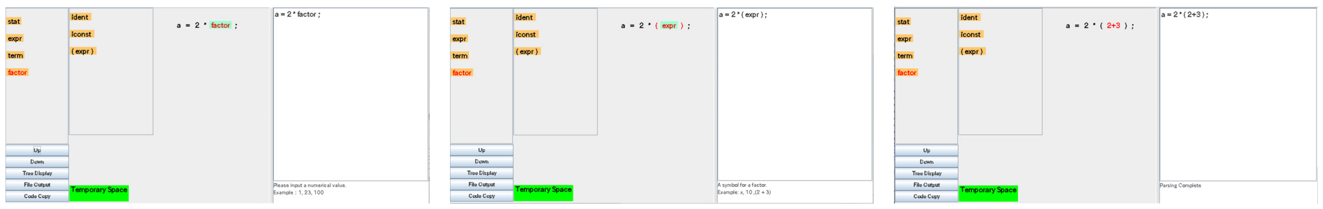

(l) Completed.

Fig. 3 Creating four arithmetic language program.



```
a = 2 * factor ;
```

Please input a numerical value.
Example : 1, 23, 100

(a) After “Up” of $\boxed{3}$ and $\boxed{\texttt{iconst}}$.

```
a = 2 * ( expr ) ;
```

A symbol for a factor.
Example: x, 10 ,(2 + 3)

(b) Replacing $\boxed{\texttt{factor}}$.

```
a = 2 * ( 2+3 ) ;
```

Parsing Complete

(c) Entering 《`expr`》.

Fig. 4 Change of four arithmetic language program.

《`iconst`》 and enter “3” (Fig. 3 (j)). Though $\boxed{\texttt{term}}$ can be derived by block operations, it is also possible to enter the content to replace the symbol by a text operation. In Fig. 3 (k), 《`term`》 is replaced by directly entering “2” from the keyboard.

Since all blocks in Fig. 3 (l) are colorless, we can see that the program creation has been completed.

It is possible to undo the derivation just made and return to the left-hand side of the syntax rule applied. For example, consider the case where we want to change the right operand “3” of 《`*`》 in Fig. 3 (l) to “`(2 + 3)`.” **Figure 4** shows this changing process. First, we select the block $\boxed{3}$ and then click the “Up” function button. Then the selected block $\boxed{3}$ is replaced with the block $\boxed{\texttt{iconst}}$, which was the one before derivation. Similarly, by clicking “Up,” the block $\boxed{\texttt{iconst}}$ is replaced with the block $\boxed{\texttt{factor}}$ before derivation (Fig. 4 (a)). After that, we replace $\boxed{\texttt{factor}}$ with $\boxed{\texttt{(}}$ $\boxed{\texttt{expr}}$ $\boxed{\texttt{)}}$ (Fig. 4 (b)), and then enter “`2 + 3`” for 《`expr`》 in Text Area. Now we have completed the change (Fig. 4 (c)).

Once program creation has been completed, it is possible to save the program to an external file by pressing the “File Output” function button. It is also possible to copy the entire program to the clipboard by pressing the “Code Copy” button.

3.5 Example: Tiny μPlan

Tiny μPlan is a simple programming language based on Micro Plan [16], a small Pascal-like language developed on the 8080 series microcomputers in 1977. **Figure 5** shows the syntax definition of Tiny μPlan in BNF.

```
⟨program⟩ ::= ⟨gdecl⟩ ⟨noemptystmt⟩ . | ⟨noemptystmt⟩ .
⟨gdecl⟩ ::= var ⟨variables⟩ ;
⟨variables⟩ ::= ⟨ident⟩ | ⟨variables⟩ , ⟨ident⟩
⟨ident⟩ ::= an alphanumeric string beginning with an alphabetic character
⟨stmt⟩ ::= (empty) | ⟨noemptystmt⟩
⟨noemptystmt⟩ ::= ⟨assignstmt⟩ | ⟨compoundstmt⟩ | ⟨ifstmt⟩ | ⟨whilestmt⟩
                  | ⟨putcstmt⟩ | ⟨putsstmt⟩ | ⟨putdstmt⟩
⟨assignstmt⟩ ::= ⟨ident⟩ = ⟨expr⟩
⟨compoundstmt⟩ ::= begin ⟨stmts⟩ end
⟨stmts⟩ ::= ⟨stmt⟩ | ⟨stmt⟩ ; ⟨stmts⟩
⟨ifstmt⟩ ::= if ⟨expr⟩ then ⟨noemptystmt⟩ else ⟨noemptystmt⟩
           | if ⟨expr⟩ do ⟨noemptystmt⟩
⟨whilestmt⟩ ::= while ⟨expr⟩ do ⟨noemptystmt⟩
⟨putcstmt⟩ ::= putc(⟨expr⟩)
⟨putsstmt⟩ ::= putc(⟨string⟩)
⟨putdstmt⟩ ::= putd(⟨expr⟩)
⟨expr⟩ ::= ⟨simpleexpr⟩ | ⟨simpleexpr⟩ ⟨relop⟩ ⟨simpleexpr⟩
⟨relop⟩ ::= : | # | < | >
⟨simpleexpr⟩ ::= ⟨term⟩ | ⟨addop⟩ ⟨term⟩ | ⟨simpleexpr⟩ ⟨addop⟩ ⟨term⟩
⟨addop⟩ ::= + | - | or
⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ ⟨mulop⟩ ⟨factor⟩
⟨mulop⟩ ::= * | div | mod | and
⟨factor⟩ ::= ⟨constant⟩ | ⟨ident⟩ | ( ⟨expr⟩ ) | not ⟨factor⟩
           | ⟨getcexpr⟩ | ⟨getdexpr⟩
⟨constant⟩ ::= ⟨int⟩ | ⟨char⟩
⟨getcexpr⟩::= getc()
⟨getdexpr⟩ ::= getd()
⟨int⟩ ::= a sequence of digits
⟨char⟩ ::= a character preceded by '
⟨string⟩ ::= a sequence of characters enclosed by " "
```

Fig. 5 Syntax of Tiny μPlan.

Here, we show the steps to create the following program for Tiny μPlan.

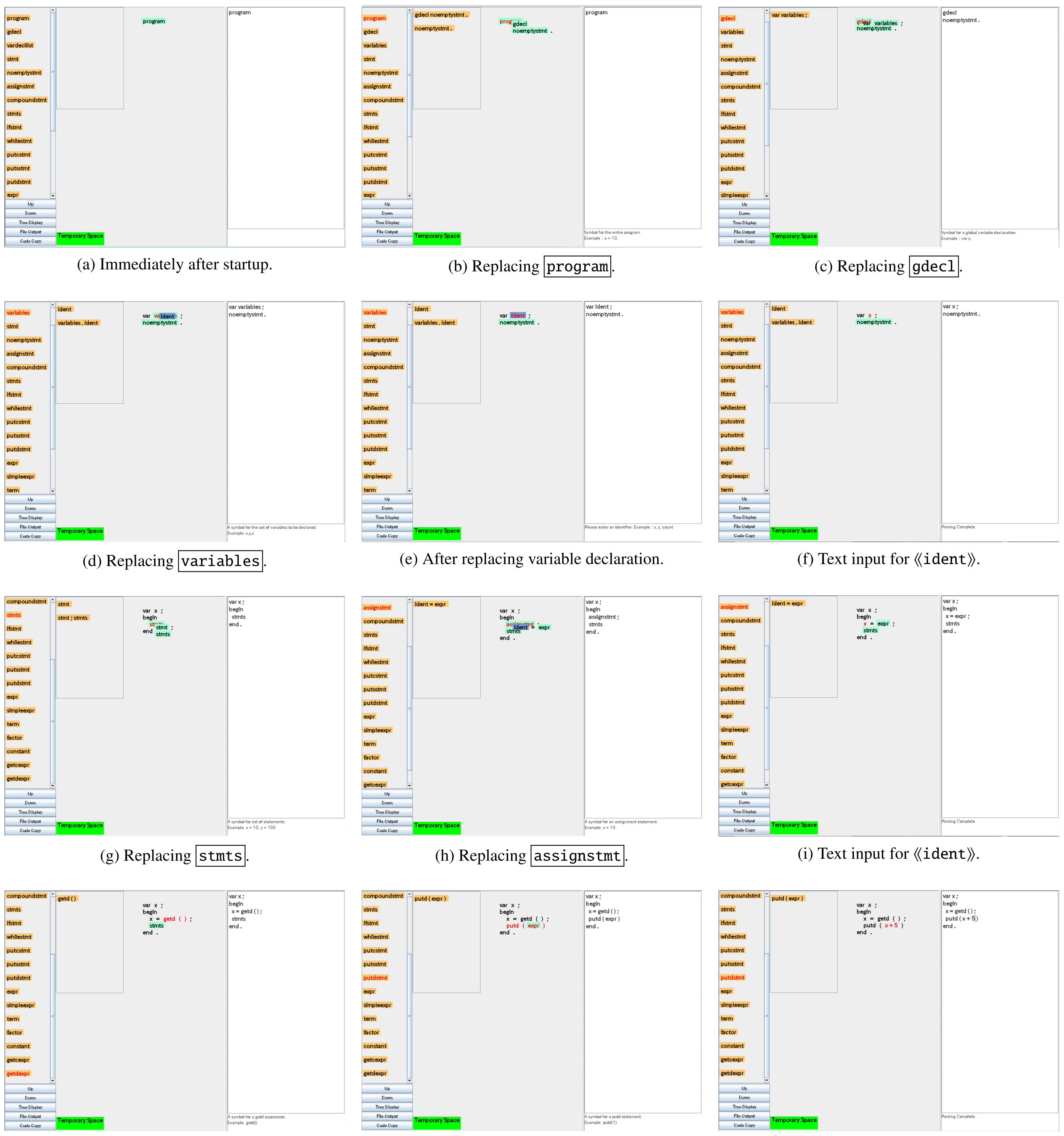
```
var x ;          // variable declaration
begin
  x = getd() ;   // input
  putd(x + 5)    // output
end.
```

This program reads the value of variable `x` and outputs the result of “`x + 5`.” It first declares the variable and describes its processing between `begin` and `end`, in which “`x = getd()`” represents the input for variable `x` and “`putd(x + 5)`” represents the output. It uses “`;`” between statements and ends with “`.`” after `end`.

The system startup screen is shown in **Fig. 6** (a). From here, the process for creating the program is shown from Fig. 6 (b) to Fig. 6 (k), and the completed program is shown in Fig. 6 (l).

First, selecting $\boxed{\texttt{program}}$ in Blocks List displays a Block Sequences List that can be replaced. By dragging the sequence $\boxed{\texttt{gdecl}}$ $\boxed{\texttt{noemptystmt}}$ $\boxed{\texttt{.}}$ out of the displayed sequences and then dropping it onto the block $\boxed{\texttt{program}}$ in Block Area, we can replace $\boxed{\texttt{program}}$ with $\boxed{\texttt{gdecl}}$ $\boxed{\texttt{noemptystmt}}$ $\boxed{\texttt{.}}$ (Fig. 6 (b)). At the same time, the symbol 《`program`》 in Text Area is replaced with 《`gdecl noemptystmt .`》. Doing similar operations on $\boxed{\texttt{gdecl}}$ (Fig. 6 (c)) and $\boxed{\texttt{variables}}$ (Fig. 6 (d)) leads to a block code presented in Fig. 6 (e), in which $\boxed{\texttt{var}}$, $\boxed{\texttt{;}}$, and $\boxed{\texttt{.}}$ correspond to terminal symbols. Since further replacements by block and text operations are impossible for terminal symbols, they are displayed colorless. $\boxed{\texttt{ident}}$ is displayed in blue, which means that it cannot be replaced by a block operation.

Next, by clicking the symbol 《`ident`》 on Text Area, the symbol is selected and the corresponding block on Block Area turns to red. Then, by typing “`x`” in Text Area, the code contents of both the block code and text code are synchronously updated. After that, by pressing the “Enter” key, the content of the input is



(a) Immediately after startup. (b) Replacing $\boxed{\texttt{program}}$. (c) Replacing $\boxed{\texttt{gdecl}}$.

(d) Replacing $\boxed{\texttt{variables}}$. (e) After replacing variable declaration. (f) Text input for 《ident》.

(g) Replacing $\boxed{\texttt{stmts}}$. (h) Replacing $\boxed{\texttt{assignstmt}}$. (i) Text input for 《ident》.

(j) Text input for 《expr》. (k) Replacing $\boxed{\texttt{stmts}}$. (l) Text input for 《expr》.

Fig. 6 Creating Tiny μPlan program.

confirmed. This input follows the syntax, so the code replacement for 《ident》 is completed (Fig. 6 (f)).

Similar to the case of editing variable declarations, we replace $\boxed{\texttt{noemptystmt}}$ with $\boxed{\texttt{compoundstmt}}$ and $\boxed{\texttt{compoundstmt}}$ with $\boxed{\texttt{begin}}$ $\boxed{\texttt{stmts}}$ $\boxed{\texttt{end}}$. Here, $\boxed{\texttt{stmts}}$ represents a sequence of statements. We replace it with $\boxed{\texttt{stmt}}$ $\boxed{\texttt{;}}$ $\boxed{\texttt{stmts}}$ to cut out a statement (Fig. 6 (g)). It is necessary to generate an assignment statement that assigns a value to variable x; we replace $\boxed{\texttt{stmt}}$ with, via $\boxed{\texttt{noemptystmt}}$ and $\boxed{\texttt{assignstmt}}$, $\boxed{\texttt{ident}}$ $\boxed{\texttt{=}}$ $\boxed{\texttt{expr}}$ (Fig. 6 (h)). Here $\boxed{\texttt{ident}}$ corresponds to variable x. Thus, we enter "x" for 《ident》 from the keyboard in Text Area (Fig. 6 (i)). $\boxed{\texttt{expr}}$ corresponds to an expression for reading a value; we replace it with, via $\boxed{\texttt{simpleexpr}}$, $\boxed{\texttt{term}}$, $\boxed{\texttt{factor}}$, and $\boxed{\texttt{getdexpr}}$, $\boxed{\texttt{getd}}$ $\boxed{\texttt{(}}$ $\boxed{\texttt{)}}$ by repeating drag-and-drop operations in Block Area. We can also enter "getd()" directly for 《expr》 in Text Area without using drag-and-drop operations (Fig. 6 (j)).

The only remaining code is the part that outputs "x + 5." To do so, we first replace $\boxed{\texttt{stmts}}$ with $\boxed{\texttt{stmt}}$, and then further replace it with $\boxed{\texttt{putd}}$ $\boxed{\texttt{(}}$ $\boxed{\texttt{expr}}$ $\boxed{\texttt{)}}$ via $\boxed{\texttt{noemptystmt}}$ and $\boxed{\texttt{putdstmt}}$ (Fig. 6 (k)). Since $\boxed{\texttt{expr}}$ is the output content, we replace it via $\boxed{\texttt{simpleexpr}}$ with $\boxed{\texttt{simpleexpr}}$ $\boxed{\texttt{addop}}$ $\boxed{\texttt{term}}$. Then we perform appropriate block and text operations, replacing $\boxed{\texttt{simpleexpr}}$ with $\boxed{\texttt{x}}$, $\boxed{\texttt{addop}}$ with $\boxed{\texttt{+}}$, and $\boxed{\texttt{term}}$ with $\boxed{\texttt{5}}$. We can also do this by typing "`x + 5`" for ⟪`expr`⟫, as in the previous example (Fig. 6 (l)).

The program has been now complete because all the blocks are colorless, which means that they have been edited. The entire code can be saved to a specified file in the same way as in the four arithmetic language example.

4. Implementation

This section describes the overall structure of TABLET and implementation details.

4.1 Overall Structure of TABLET

By letting L be a target text-based language, **Fig. 7** presents the overall structure of TABLET. We used Java as the implementation language. In this figure, yellow-lined parts represent the programs and files developed in this research, and blue-lined parts indicate existing programs and automatically-generated files by, e.g., a compiler.

The instructor who uses TABLET prepares a BNF for the target language L in accordance with the description rules of SableCC [9], [10]. Since SableCC adopts LALR(1), L's grammar is subject to this restriction. However, we accepted this restriction because L is, as discussed in Section 1, supposed to be a less complex language as a destination of the shift from block-based languages. An L's parser can be obtained by giving this BNF to SableCC. In addition, the BNF of L is passed through the syntactic elements retrieval program developed in this research to create syntactic elements data of L handled by TABLET. These are combined with Java programs for the basic structures of TABLET, such as text and block operation parts, to construct TABLET specialized to L. If the target text-based language is another one L', we can obtain TABLET for L'. The learner makes a program using TABLET and saves the completed program to an external text file.

4.2 Making Code by Block and Text Operations

The code in Block Area should always be a sequence of symbols in accordance with the BNF. This is because if blocks can be freely inserted into the code, the code might contain syntax errors. Thus, we designed TABLET so that blocks taken from the Block Sequences List cannot be inserted freely into the code; the only way to create code in Block Area is, on the basis of the derivation, by dragging a block sequence from the Block Sequences List and dropping it on a block of the same syntax element to replace in the existing code. A textual code fragment is given by selecting a symbol to be replaced by text input in Text Area, and then providing input for it from the keyboard.

The code that has been created or in the process of being created by the block operations is kept as a concrete syntax tree inside the TABLET system. The concrete syntax tree grows downwards by the learner's derivation operations. The "Undo" operation described in Section 3.3.5, which cancels the derivation and goes back to the left-hand side of the syntax rule just applied, is implemented by going up the syntax tree to the root direction.

To input textual code in Text Area, the learner selects the target symbol on Text Area and enters the text to be replaced using the keyboard. After entering text, the learner's pressing the "Enter" key causes the parser generated by SableCC to parse the input. If the parsing is successful, the input is confirmed and the replaced symbol is updated to the input inside the system. If the input violates the syntax rules, the parsing will fail. In this case, an error message is displayed in Text Log to inform the learner, and symbols on the text and block areas are not updated because the input is not confirmed. Unfinished input is displayed in red in Text Area. If the target for the current operation moves to another element, the pending input reverts to the content before editing.

Syntax elements that can be entered in text are not only lexical tokens but also arbitrary non-terminal symbols. The code fragment to be replaced with this non-terminal symbol can be given from the keyboard. This enables the learner to make programs with text input in accordance with the learner's achievement level of programming.

4.3 Additional Information by Comments

The BNF for TABLET includes additional information, such as amounts of indentation when displaying code in Text Area and texts for pop-up explanations of symbols, in the form of SableCC comments. For example, consider the SableCC's BNF in **Fig. 8**, which is the definition of ⟨compoundstmt⟩ of Tiny μPlan shown in Fig. 5.

A comment in the form of "`/*` x`:`y `*/`" between elements in the right-hand side of the syntax rule indicates the information for an indented display of the code. Here, x is the line break priority, for which the smaller the value, the higher the priority, and y is the amount of indentation. When the number of blocks exceeds a predetermined number in the horizontal direction, line breaks are performed at high priority locations. Priority 0 means that line breaks are always to be performed regardless of the number of blocks. When a line break is performed, the y at this position is used for the number of spaces inserted for indentation. If y

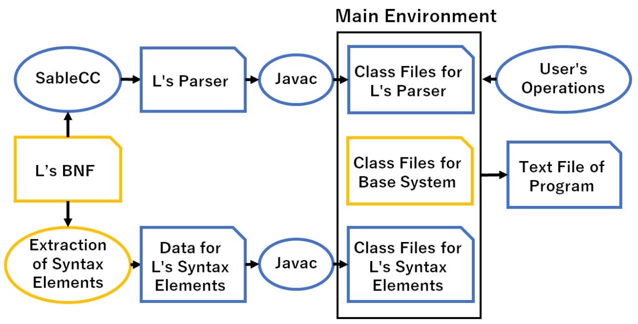

Main Environment
SableCC
L's Parser
Javac
Class Files for L's Parser
User's Operations
L's BNF
Class Files for Base System
Text File of Program
Extraction of Syntax Elements
Data for L's Syntax Elements
Javac
Class Files for L's Syntax Elements

Fig. 7 Overall structure of TABLET.

```
compoundstmt = {one} begin /* 0:2 */ stmts /* 0:-2 */ end
           ;
/* replace: {begin a = 1 end} */
/* hint: Japanese "description in Japanese",
        English  "compound statement" */
/* example: Japanese "explanation and example in Japanese"
           English  "Symbol for compound statement.
                    Example: begin x = 10; y = 100 end" */
```

Fig. 8 Syntax rule with additional information.

is a negative value, the amount of indentation is reduced by y. In the example of Fig. 8, "`begin /* 0:2 */ stmts /* 0:-2 */ end`" for `compoundstmt` represents the sequence "`begin stmts end`," which specifies that line breaks are definitely required both between `begin` and `stmts`, and between `stmts` and `end`. This also specifies that `stmts` is indented by two spaces, and `begin` and `end` are not indented.

A comment surrounded by "`/* replace: {`" and "`} */`" is the "replacement text" for the symbol, which are used in parsing a half-finished code. When parsing, it is necessary to give a complete code without any uninstantiated non-terminal symbols to the parser. Thus, to judge whether or not a half-finished code that contains such non-terminal symbols is syntactically correct by the parser, it is necessary to replace every non-terminal symbol left with a concrete text that is correct from the viewpoint of the parser. Thus, TABLET passes the code after replacing every non-terminal symbol with its corresponding replacement text specified in the comment so as not to induce a parser error. In this example, if the code contains uninstantiated symbol `compoundstmt`, TABLET replaces it with "`begin a = 1 end`" and invokes the parser. Note that it is sufficient that a replacement text is syntactically correct; it is not necessary to consider its meaning.

A comment surrounded by "`/* hint:`" and "`*/`" is the "popup information" for the syntax element, which pops up and is displayed when the mouse hovers over the symbol in the block code. The pop-up information presents a simple explanation of the symbol to the learner.

A comment surrounded by "`/* example:`" and "`*/`" is the explanation with a concrete example for the syntax element. These are displayed in Text Log when the symbol of the syntax element is selected in Text Area.

For both the pop-up information and the explanation with a concrete example, `Japanese` and `English` indicate the natural languages supported by TABLET, and the following content enclosed in double quotations is displayed in each natural language. In Fig. 8, the pop-up information and the explanation with a concrete example in English are "`compound statement`" and "`Symbol for compound statement. Example: begin x = 10; y = 100 end`," respectively.

As previously described, TABLET requires additional information about blocks to be provided as annotations in the syntax rules of SableCC. This can be a burden for those who write the syntax rules of SableCC. Reducing this burden is left for our future work.

5. Experiments for Evaluation

This section describes the details and obtained results of evaluation experiments of TABLET with testee students. Experiments were conducted when the following two features, "temporary saving of code fragments" (Section 3.3.7) and "switching display language" (Section 3.3.8), were not yet implemented. We used Tiny μPlan described in Section 3.5 for the target text-based language because it was totally new for the testee students.

5.1 Purpose of Experiments

The purpose of the experiments was to investigate the usability of TABLET and the learners' comprehension of the target text-based language. Through these experiments, we performed objective evaluations of TABLET.

Table 1 Testee's programming experiences.

No.	Text-based language		Block-based language		Others
	Name	Length	Name	Length	
1	C	2 yrs			
	C++	2 yrs			
	Java	6 mths			
	Lisp	6 mths			
2	C#	2 yrs			*1
	C	1 yr			
3	Python	2 yrs	Scratch	1 yr	
	C	1 yr			
	Ruby	1 yr			
4	C	2 yrs			*2
	C#	1 yr			
	C++	6 mths			
	Ruby	6 mths			
5	C#	2 yrs	Scratch	1 mth	
	C	1 yr			
	C++	3 mths			
	Ruby	3 mths			
6	C	1 yr			
7	C	1 yr			
	Ruby	6 mths			
8	C	7 yrs	Micro:bit	1 mth	*3
	JavaScript	3 yrs			
	C++	4 yrs			

*1: Game programming by Unity.
*2: Coding while reading programming books.
*3: Creating plug-ins.

5.2 Testees

Testees were eight second-year undergraduate students in the authors' affiliated university, who attended the lecture entitled "Introduction to Programming" held in the first semester of FY2021. This lecture was an introductory programming course for beginners by using the C programming language.

Since compulsory programming education in the elementary school started from 2020 in Japan, it was difficult to find beginner students who had learned block-based languages. Thus, the testee students previously described were appointed in the experiments.

We assigned unique numbers for all testees and used those numbers to identify individual students without using their real names.

Table 1 shows the eight students' programming experiences. With block-based languages, the maximum experience was one year, and five students had no experience. In contrast, students had at least one year of experience with text-based languages. However, all students had no experiences with Tiny μPlan at all.

5.3 Methods of Experiments

We conducted the experiments remotely by using Zoom, with each student at a location with a good network environment. The experiments consisted of three sessions, each of which took about 90 minutes. The experiments were scheduled at each student's convenience. The contents of each session were as follows.

Session 1 Explanations of TABLET and Tiny μPlan, and programming with simple inputs and outputs.

Session 2 Programming with control structures (`if` and `while` statements).

Session 3 Creation of rather complex programs.

At the beginning of each session, the first author of this paper gave an oral explanation on the basis of handouts given. The testee students then created their assignment programs on TABLET and submitted their programs. When submitting a program, each student ran the program through an execution form on a dedicated Web page to check if the result was correct. The student filled the form with the program code and, if any, the input given from the standard input. After submitting the program, the student were asked to answer a questionnaire.

All executions of submitted programs were recorded on the server. In addition to information such as the submission status and the time taken to create the program, information from the questionnaire was used in the evaluation of TABLET.

The assignment programs in each session were as follows. Compared with the assignment programs in the first two sessions, those in Session 3 were longer and more complex.

Session 1 (Input and output)

- A program that outputs the string "`Hello World`."
- A program that reads two integers x and y, and outputs the results of $x + y$, $x - y$, $x \times y$, and x/y.

Session 2 (Control structure)

- A program that reads an integer x and outputs "`Positive`" if positive, "`Negative`" if negative, or "`Zero`" if zero.
- A program that reads two positive integers x and y, and outputs x^y.

Session 3 (Summary)

- A program that reads three integers and outputs their median.
- A program that reads two integers and outputs their greatest common divisor.
- A program that reads an integer and outputs its prime factorization.

The questionnaire asked the following questions about TABLET on a scale out of five.

- Ease of creating programs by using block operations.
- Ease of creating programs by using text operations.
- Ease of creating programs by using both block and text operations.
- Ratio of block and text operations used.
- Ease of understanding the correspondence between block and text code.
- Comprehension level of the syntax of the target text-based language, Tiny μPlan.

5.4 Experimental Results

5.4.1 Assignments

In both Sessions 1 and 2, no student failed to make programs for the assignment problems. In Session 3, however, two students exceeded the time limit during the third problem.

When executing the submitted program, there were no cases in which any syntax error occurred. We believe that this is because, in the first place, TABLET creates programs on the basis of the derivation operations of blocks; it is essentially impossible to create programs that are against the syntax of the text-based language. Two other possible reasons are that the block color made it easy for the student to identify unedited elements, and that block operations automatically supply terminal symbols such as semicolon, which are often forgotten in text-based programming.

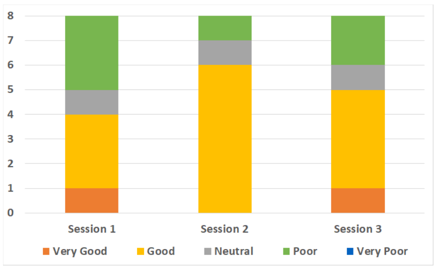


Fig. 9 Ease of block operations.

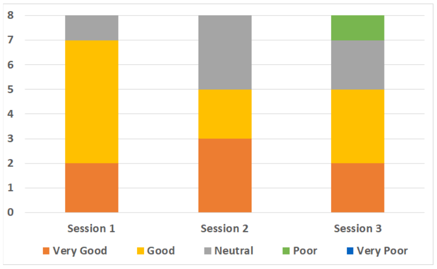


Fig. 10 Ease of text operations.

5.4.2 Questionnaire Results

Figure 9 shows the ease of creating programs by using block operations. The percentage of "Good" or "Very good" in Session 1 was smaller than those in Sessions 2 and 3. This is presumably because most of the students had no programming experience in block-based languages, and were therefore bewildered by their first use of a block-based language. In addition, it might be difficult to understand the meanings of blocks at a glance because they were named after the syntax rules.

In Session 2, six students selected "Good," which increased compared with Session 1. We believe this is due to the fact that the students became accustomed to the block operations through Sessions 1 and 2 and were able to grasp the meanings of blocks through Text Log and pop-up descriptions.

However, the percentages of "Good" or "Very good" decreased in Session 3 compared with Session 2. Two reasons are considered for this. First, programs in Session 3 were more complex and longer than those in Sessions 1 and 2. Second, TABLET at the time of the experiments was not equipped with the feature of temporarily savings code fragments; to insert a new code fragment in the middle of the already-created program, it was necessary to go up to the inserting position, insert the code fragment, and then create the same code as before from the first.

Figure 10 shows the ease of creating programs by using text operations. Seven students chose "Good" or "Very good" in Session 1, presumably because all students had experiences of text-based languages. Despite this, the number of students who selected "Good" or "Very good" in Sessions 2 and 3 decreased a little compared with Session 1. This was the opposite result of Fig. 9; this might be due to the fact that the students began to become accustomed to block operations and felt that block operations were easier than text operations.

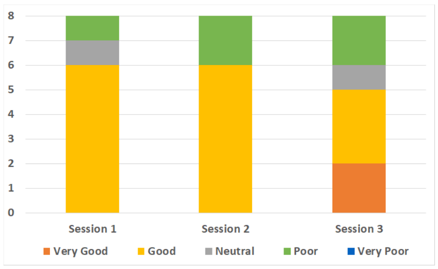


Fig. 11 Ease of block and text operations.

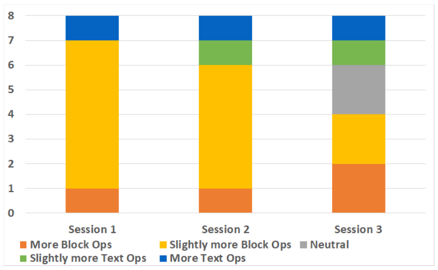


Fig. 12 Ratio of block and text operations.

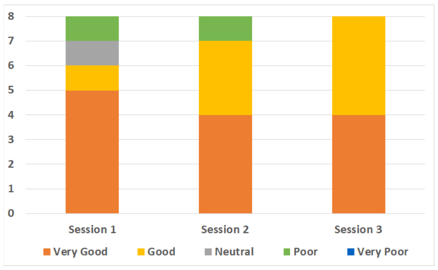


Fig. 13 Ease of understanding the correspondence.

Figure 11 shows the ease of creating programs by using both block and text operations. The number of students who answered "Very good" increased in Session 3. This is presumably because students became familiar with the operations in TABLET through Sessions 1 and 2, and, as a result, code fragments that should be created with block operations and those that should be created with text operations became clearer.

Figure 12 shows the ratios of block operations and text operations used. We can see that more block operations than text operations were used in Sessions 1 and 2. This is presumably because programs were short and simple, so block operations were less complicated with fewer operations. The percentage of block operations decreased in Session 3. Since programs in Session 3 were more complex and longer, depending on too many block operations might increase the amount of time and effort, and induce misoperations; we believe that the number of cases in which the testee students relied on text operations increased for these reasons.

Figure 13 shows the ease of understanding the correspondence between block and text code. The percentage of "Good" and "Very good" monotonically increased from Session 1 to 3. From these results, it can be considered that TABLET was effective in facilitating the understanding of the correspondences between block and text code.

Finally, **Fig. 14** shows the comprehension level of the syntax of Tiny μPlan. The responses "Good" and "Very good" accounted

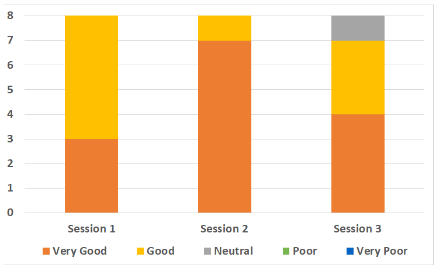


Fig. 14 Comprehension level of the syntax.

for 100% for both Sessions 1 and 2. From this result, we believe that TABLET helps the learner understand the syntax of the target text-based language. This evaluation is also supported from the fact that all students were able to create the assignment programs in Sessions 1 and 2. In addition, from Fig. 13, the correspondence between block and text code was presumably easier to grasp for the testee students.

However, the percentage decreased in Session 3. This might be because experimental time for using TABLET was short for several testee students. In fact, two students were not able to finish the assignment programs within the time limit.

6. Related Work

6.1 Structural Editors and their Generators

HASKEU [17] is a Haskell programming development environment that supports both visual programming based on a structural editor and text-based programming. It is equipped with a propagation mechanism of program changes bidirectionally to maintain the consistency between them. Users can learn programming quickly by using visual programming, and when they become ready, they can move on to a more advanced level by using the text-based programming. The target text-based language of HASKEU is restricted to a functional language Haskell; HASKEU does not support the generality like TABLET, which can cope with many languages.

For systems that are capable of automatically generating structural editors from the syntax definitions of target languages, many studies including the system by Arefi el al. [11], ASF+SDF Meta-environment [12], LISA [13], and MPS [14], have been conducted. Ferreira's syntax-directed editor generator [15] is more closely related to TABLET because it generates, given the syntax definition, a structural editor that enables both syntax-directed editing through visual operations and text-based editing. This structural editor consists of two parts: a visual representation of the program's syntax tree and a textual representation of the program. Syntax-directed editing is done by mouse operations on the syntax tree. Text-based editing is done by replacing tokens such as identifiers and numerical literals with their concrete contents. When terminal symbols and/or tokens appear during the editing process of the syntax tree, they also appear in the textual representation. When the user enters their concrete contents in the textual part from the keyboard, the other part reflects the content; real-time synchronization similar to that of TABLET works. However, it is impossible to enter texts for non-terminal symbols in the middle of the syntax tree, which is possible in TABLET. Thus, the user cannot adjust the range of syntax elements for which text inputs are possible in accordance with the learning stage.

6.2 Block-oriented Systems

Scratch [1], [2], [3] is a language for creating programs by snapping together blocks with various functions. It has been designed, developed, and maintained by MIT Media Lab Lifelong Kindergarten Group. Blocks have notches and bumps, and the user creates a program by fitting desired blocks together. There are no textual representation for a program; only a block representation is displayed in the Scratch window. According to the Scratch home page [1], Scratch is used in more than 200 countries and regions, and supports more than 70 languages.

As a practical example in elementary school education, Mori et al. [18] used Scratch to teach a class of 38 students in the fourth grade. The content of the class was designed to create programming works. More than 80% of the children were able to work on making programs, which included conditional branches and so on. In the class, Scratch was highly rated in terms of children's interest in programming. This indicates that block-based languages such as Scratch are expected to be very effective in elementary school education, and many children are expected to learn block-based languages in the near future in Japan. Target users of TABLET are learners who have mastered programming in block-based languages.

Blockly [4], [5] is a block-based language developed by Google. A program created by using blocks can be textually displayed as a program in text-based languages such as JavaScript, Python, etc. in real time. The character display on a block is based on natural language, which is different from the code notation of a text-based language. However, since every element of the block and that of the text correspond to each other, it is easy for the user to correlate the contents of the two.

Although Blockly can display block code in text-based languages, it is impossible to edit the textual code directly; only one-way conversion from block to text is possible. The user can understand the correspondences between block code and text code, but cannot give a textual program in terms of the syntax of a text-based language.

From the viewpoint of the shift from a block-based language to a text-based language, there might be an opinion that, a system like Blockly that enebles only block-based programming and presents textual code at the same time would suffice for a user to grasp the correspondences between both code and then to shift to text-based programming. However, we believe that such passive learning, in which the user only reads the textual code presented by the system, is insufficient to learn how to write programs in a text-based language, even if the user can feel the flavor of text-based programming. Thus, though partially, TABLET adopts the user's active text-based programming to provide the user with the experience of writing textual code directly, and aims for a smooth shift to a text-based language.

Matsumoto et al. proposed OCaml Blockly [*4], a programming environment of OCaml that enables intuitive programming by using blocks. OCaml Blockly was designed to let the user learn the OCaml language specification through block-based programming and then move to text-based programming. To this end, conversion between block code and textual OCaml code was possible.

Conversions between block and text code are not done in real time; the user has to click the "conversion button" every time conversion is necessary. It is therefore rather difficult to know which part of the block code corresponds to which part of the textual code. This could be a potential problem of OCaml Blockly in the shift to text-based programming.

[*4] http://pllab.is.ocha.ac.jp/~asai/jpapers/ppl/matsumoto19.pdf (in Japanese) (accessed 2022-06-07)

6.3 Shift to Text-based Language Support

Playgram [19] is a programming learning tool developed by Preferred Networks Inc. Starting with visual programming, it supports step-by-step learning, i.e., typing, basic programming, and textual coding in Python. By using 3D graphics, Playgram aims to help the user acquire the ability to freely express themselves and recognize space while solving problems by making full use of programming. As a mechanism for supporting the user's shift from a visual language to a text-based language, Playgram converts the textual program into a program in the visual language and then executes the program so that their execution results are always the same. In addition, in accordance with the learning stage of each user, Japanese displays on blocks can be replaced with Python-based displays. Different from TABLET, Playgram does not provide support for various text-based languages.

7. Conclusion

In this paper, we proposed TABLET, a programming environment that supports the shift from a block-based language to a text-based language for block-based language learners. We have designed TABLET as a syntax-directed system that focuses on making learners aware of the target text-based language's syntax. Concretely, TABLET aims to integrate both block-based programming and text-based programming by adopting both block operations using a mouse and text operations using a keyboard. To achieve generality for a variety of text-based languages, TABLET is designed as a system that is capable of generating blocks on the basis of a target text-based language's BNF. Through evaluation experiments with testee students, TABLET is shown to be potentially useful in the learners' shift from block to text-based languages.

In the current TABLET, repetition of the same non-terminal symbol must be recursively defined in the BNF, as for the ⟨term⟩ in Fig. 2. Extending TABLET to introduce repetition symbols like extended BNF and to provide special operations for them is left for our future work.

Another issue for future work is to conduct experiments with novice programmers who have only learned block-based languages. Through these experiments, it is necessary to further investigate whether they can understand the syntax-directed derivation operations of blocks, which are different from those of existing block-based programming language, and whether they can advance their understanding of the syntax of the target text-based language.

References

[1] Scratch (online), available from ⟨https://scratch.mit.edu/⟩ (accessed 2022-05-02).

[2] Maloney, J.H., Resnick, M., Rusk, N., Silverman, B. and Eastmond, E.: The Scratch Programming Language and Environment, *ACM Trans. Comput. Educ.*, Vol.10, No.4, pp.16:1–16:15 (2010).

[3] Armoni, M., Meerbaum-Salant, O. and Ben-Ari, M.: From Scratch to "Real" Programming, *ACM Trans. Comput. Educ.*, Vol.14, No.4, pp.25:1–25:15 (2015).

[4] Blockly (online), available from ⟨https://developers.google.com/blockly/⟩ (accessed 2022-05-02).

[5] Seraj, M., Katterfeldt, E., Bub, K., Autexier, S. and Drechsler, R.: Scratch and Google Blockly: How Girls' Programming Skills and Attitudes are Influenced, *Proc. 19th Koli Calling International Conference on Computing Education Research* (*Koli Calling 2019*), pp.23:1–23:10 (2019).

[6] Viscuit (online), available from ⟨https://www.viscuit.com/⟩ (accessed 2022-05-02).

[7] Harada, Y. and Potter, R.: Fuzzy Rewriting – Soft Program Semantics for Children, *Proc. 2003 IEEE Symposium on Human Centric Computing Languages and Environments* (*HCC 2003*), pp.39–46 (2003).

[8] Watanabe, T., Nakayama, Y., Harada, Y. and Kuno, Y.: Analyzing Viscuit Programs Crafted by Kindergarten Children, *Proc. 2020 ACM Conference on International Computing Education Research* (*ICER 2020*), pp.238–247 (2020).

[9] Gagnon, É.: SableCC, An Object-Oriented Compiler Framework, School of Computer Science, McGill University (1998).

[10] SableCC (online), available from ⟨https://sablecc.org/⟩ (accessed 2022-05-02).

[11] Arefi, F., Hughes, C.E. and Workman, D.A.: Automatically Generating Visual Syntax-Directed Editors, *Comm. ACM*, Vol.33, No.3, pp.349–360 (1990).

[12] van den Brand, M., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E. and Visser, J.: The ASF+SDF Meta-environment: A Component-Based Language Development Environment, *Proc. 10th International Conference on Compiler Construction* (*CC 2001*), Lecture Notes in Computer Science 2027, pp.365–370 (2001).

[13] Henriques, P.R., Pereira, M.J.V., Mernik, M., Lenic, M., Gray, J. and Wu, H.: Automatic Generation of Language-based Tools using the LISA System, *IEE Proc. Softw.*, Vol.152, No.2, pp.54–69 (2005).

[14] Voelter, M. and Pech, V.: Language Modularity with the MPS Language Workbench, *Proc. 34th International Conference on Software Engineering* (*ICSE 2012*), pp.1449–1450 (2012).

[15] Ferreira, J.M.S.: Syntax-Directed Editor Generator, University of Minho (2017).

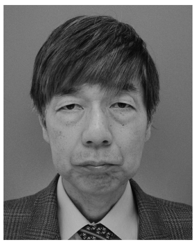
[16] Ishida, H.: Microcomputer Languages, *Journal of Information Processing Society of Japan*, Vol.22, No.6, pp.501–504 (1981) (in Japanese).

[17] Alam, A. and Bush, V.: HASKEU: An Editor to Support Visual and Textual Programming in Tandem, *Proc. 2016 SAI Computing Conference*, pp.805–814 (2016).

[18] Mori, H., Sugisawa, M., Zhang, H. and Maesako, T.: Practical Study on Scratch Programming Lessons for Elementary School Students, *Japan journal of educational technology*, Vol.34, No.4, pp.387–394 (2011) (in Japanese).

[19] Playgram (online), available from ⟨https://playgram.jp/⟩ (accessed 2022-05-02).

Takumi Miyajima received his B.E. and M.E. degrees from the University of Electro-Communications in 2020 and 2022, respectively. He is currently working in PCI Solutions INC. His research interests are visual programming languages and systems, and systems software.



Hideya Iwasaki is a professor in the School of Science and Technology at Meiji University, Japan. Until the end of March 2022, he had been a professor in the Graduate School of Informatics and Engineering at the University of Electro-Communications. He has been a member of the Science Council of Japan since 2011. He received an M.E. degree in 1985 and Dr.Eng. degree in 1988 from the University of Tokyo. His research interests includes programming languages and systems, parallel programming, systems software, and constructive algorithmics. He is a member of the IPSJ and ACM.

Yasushi Kuno is a Professor Emeritus of University of Tsukuba, Japan. He received his B.S., M.S., and D.Sci. degrees from Tokyo Institute of Technology in 1979, 1981, and 1986, respectively. He was an Assistant Professor of Tokyo Institute of Technology from 1984 to 1989, a Lecturer, an Associate Professor, and a Professor of University of Tsukuba from 1989 to 2016, and a Professor of the University of Electro-Communications from 2016 to 2022.