International Journal of Learning Technologies and Learning Environments
International Institute of Applied Informatics
2020, Vo. 3, No. 1, 35 – 50

Dealing with Stumbling in C Language Programming Using Visual Programming Environment

Kousuke Abe [*], Yuki Fukawa [*], Tetsuo Tanaka [*]

Abstract

For the education of beginning programmers, visual programming that develops programs by combining blocks has attracted significant attention. An environment for generating code in a conventional programming language is also provided. However, existing environments are not fully visualized. In this investigation, we prototyped a development environment for the C language in which users can intuitively understand the concept of variable declarations and include statements, and an execution environment that visualizes the state of evaluation of expressions and changes in the values of variables before and after the execution of the statement. It also has step-forward and step-backward functions. This programming environment is a web application developed with JavaScript. For step-by-step evaluation of an expression, it converts the expression internally to Reverse Polish Notation and visualizes the change in the terms in the expression. To implement the step-backward function, it has a history-of-execution context. We determined experimentally that students who are not proficient in C can program more accurately and quickly in this environment than with text-based coding.

Keywords: visual programming, block-based, C language, novice programmer, web application, interpreter

1 Introduction

In higher education institutions, the C language is often the first programming language to be learned. This is also true in the authors' institution, the Kanagawa Institute of Technology Department of Information Engineering. C language classes are held three times per week in the first year. Students use Visual Studio for programming. However, in text-based C language programming, students must memorize keywords such as int, double, return, break, or the syntax of an if statement, for statement, and so on, which places hurdles in front of beginners. Furthermore, in the case of a computer beginner, it takes time to edit a program because they are not used to keyboard input. Even if the student can finally input the program, merely including a

[*] Kanagawa Institute of Technology, Kanagawa, Japan

single typo would produce a compilation error, yielding frustration rather than positive motivation [1][2].

On the other hand, the research and development of a programming support system for novice programmers has reached an advanced stage. One development is block-based visual programming such as Scratch [3] or Snap! [4]. In block-based programming, students edit programs by selecting blocks and combining them. Thus, they can program even without remembering or formulating the grammar correctly. The effectiveness of block-based visual programming has been reported [5][6][7][8][9].

However, Scratch and Snap! use their own programming languages and are not suitable for learning popular programming languages such as C or Java. There are visual programming environments for existing programming languages like Google Blocky [10], BlockEditor [11], and Pencil Code [12], but their visualization is not adequate. For example, BlockEditor cannot execute the block-based programs as they are created. Since they output text-based source code, users have to paste and execute their source code in another development environment such as Microsoft Visual Studio. Therefore, in addition to the editor, it is necessary to prepare the execution environment of the program.

With these factors in mind, in this investigation we developed a C language visual programming environment with the aim of lowering the barriers between beginning programmers and the learning of C. This programming environment is a Web application that has an editing function to edit a C language program and an execution function that can step through the program and trace the changes in the variables being executed.

In this paper, Section 2 describes common stumbles in programming and how to deal with them, and Section 3 outlines the programming environment. Section 4 describes its implementation and Section 5 describes the results of a trial experiment.

2 How to Deal with Stumbling in Programming

2.1 Block-based Visual Programming

In programming with a general-purpose language such as C, the programmer has to remember keywords such as #include, int, float, return, break, and the syntax required for statements such as if and for. Such hurdles are high for beginners. Many novice programmers do not fully understand the basic concept of grammar, and often make simple errors [13]. Even if the error is simple, it can be difficult to identify and deal with it [14] [15]. Furthermore, beginners with PCs are not used to keyboard input, so it takes time to edit programs. Even if they can finally input a program, they will get a compile error from just one typographical error. If they make a mistake in the name or declaration of a variable or function, they get an error message saying "variable or function is not declared". But since they feel sure that they declared it, they do not know what is wrong. At that point, they stumble. Too much such frustration, and they will lose their motivation.

In this investigation we provided a visual programming environment for editing programs by combining blocks. The environment presents a list of blocks corresponding to programming actions (function declaration, assignment, branching, repetition, etc.) that are necessary for programming. This eliminates the need for the user to correctly memorize C language keywords and syntax. Also, since typing errors are eliminated, grammatical errors can be reduced. For users who type slowly, keyboard input is reduced, making it easier to program.

When the user declares variables or functions, the environment generates corresponding variable blocks or function blocks. This enables the user to select them when referring to variables or calling functions. For standard functions such as printf, the same can be done at the time of inserting the corresponding #include statement. This makes it possible to eliminate errors in variable names and function names. Users will also be able to intuitively understand the concepts "variable can be referenced after declaring" and "function can be called after declaring". In addition, the user can learn the role of the include statement.

2.1 Various Step Executions and Tracing

When learning programming, students initially come to understand concepts such as assignment, branching, and repetition. Next, they come to understand how the program works by carefully reading the sample program, absorbing the concepts one line at a time, and confirming the behavior by typing it into the computer and executing it. In addition, they will deepen their understanding by changing parts of the program or by developing programs dealing with similar problems on their own.

However, for users who are not good at programming, it is difficult to understand how a program works. They often do not understand it, but they are satisfied when they input a program according to the sample and get the expected answer. However, this makes it difficult to improve their programming skills.

Also, when a logical error occurs, it will be in a situation such as producing an erroneous answer, crashing, stopping, and so on. Logic errors differ from grammar errors: no error message is displayed, so the novice programmer cannot find out what kind of error it is, and stumbles.

To understand the behavior of the program and to debug it when a logic error is involved, step-by-step execution in the debugger and tracing the value of each variable at the time of execution are useful. However, in the step-by-step execution of a typical debugger, since it is executed on a statement-by-statement basis, the way that the value of the expression changes is not visualized.

This programming environment steps through each evaluation of the expression. For example, the assignment statement shown in Figure 1 is decomposed until the evaluation step of the expression and executed, as shown in Figure 2.

```
int x = 3;
y = x = x + 1;
```

Figure 1: Example of Assignment Statement

y = x = **x** + 1	(evaluate **x**	value of x: 3)
y = x = **3 + 1**	(evaluate **3 + 1**	value of x: 3)
y = **x = 4**	(evaluate **x= 4**	value of x: 4)
y = 4		

Figure 2: Example of Expression Step Evaluation

This makes it more intuitive for the user to follow the updating of the value by assignment, to have the value on the right side of the assignment show the value before substitution, the expression containing the value, and the assignment also presented as an expression.

Also, the conventional debugger, once there is a crash, cannot go back to the execution of the program to show the value of the variable immediately before the crash. In order to refer to that value, it is necessary to stop and execute debugging again, which is time-consuming and troublesome.

This programming environment provides the function of step-back (function to go back by one step), and it always displays the values of variables during program execution, the history of function calls, and the result of console output. In addition, the statement that is being executed, the variable whose value has changed, and the character string newly outputted to the console are highlighted.

This makes it possible to grasp the value of each variable and perceive how the console changes each time it executes one step. Also, when a crash happens, the user can step back and check the value of the variable immediately before the crash, and thus can efficiently ascertain which value is in error.

3 Overview of Programming Environment

In this section, we will describe how the user sees this programming environment, the editing function, and the execution function.

3.1 User's Image

This programming environment has program editing and execution functions. The user creates and edits the program with the editing function and executes it with the execution function. This programming environment is a single-page application consisting of a menu bar, a block list pane, a program pane, and an execution context pane, as shown in Figure 3. The user can edit a program by dragging and dropping blocks from the block list pane to the block pane, and execute it by pressing a button on the menu bar. In addition to normal execution, there are step-over, step-in, and step-back functions, and a rewind function to return to the beginning of the program.

3.2 Editing Function

The purpose of the program editing function is to create a program (block-based source code). The user creates a program by dragging and dropping a block from the block list pane to the program pane. This eliminates the need for the user to accurately recall the keywords and syntax of C. Also, since input errors are eliminated, grammatical errors can be minimized.

In the block list, the following blocks are prepared in advance.

- Include statement
- Variable declaration
- Function declaration
- Control statements (if, if-else, while, for, do-while, return, break, continue)
- Comment

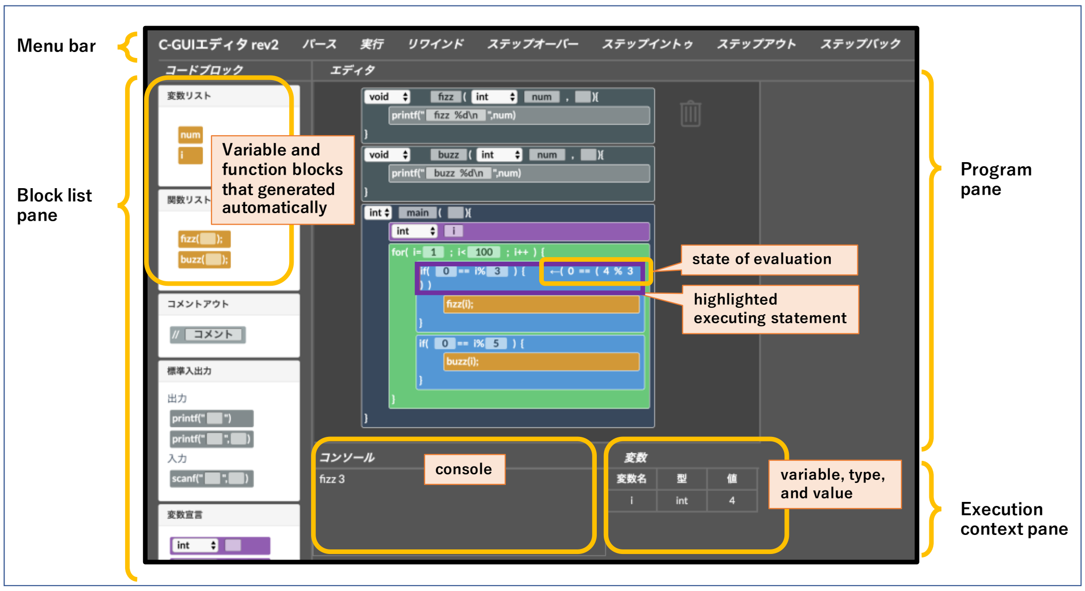

Figure 3: Appearance of the Programming Environment

Also, the following variable blocks and function blocks are dynamically generated in the block list when they are declared.

- Included built-in functions
- Declared variables
- Dummy arguments in function declaration
- Array elements of the declared array
- Declared functions

The built-in functions are displayed in the block list at the time when the #include statement is dropped into the program. Also, at the time of adding a declaration block, variables and functions are added to the block list pane as a block for reference to the variable and a function-calling block.

This allows the user to visually understand that "variables and functions are used after declaring". It also reduces errors due to mistakes in keyboard input.

A block cannot be dropped if the placement is grammatically incorrect. An error message indicating that a block cannot be placed at that location is displayed, and the block is restored. This makes it easier for users to create grammatically correct programs, and if an intention or execution is incorrect, they can immediately know the reason. Places where blocks cannot be dropped include:

- Arguments of function or operator: Blocks of a type different from the argument type
- Left side of the assignment expression: Block other than modifiable left side value
- Inside of function declaration: Function declaration block

This can reduce grammatical errors and the user can focus on examining algorithms.

3.3 Execution Function

The program execution function executes a program and displays the execution context. Here, the execution context is an overview of the state of program execution, that is, the value of the variable, the function called, the console output, and the exception occurrence situation. For each function call or expression evaluation, a function call context or an expression evaluation context is generated, respectively.

In addition to the function to execute to the end of the program, the execution environment has step-over, step-into, and breakpoint-setting/cancellation functions as in a conventional debugger. In addition, it has a step-back function that returns to the situation before execution of the sentence.

In the step-in function, step execution is performed for each expression evaluation, not for each statement execution. As a result, the user can intuitively grasp the behavior of the program, and when there is a logical error, it is easy to specify the error.

4 Implementation of This Programming Environment

4.1 Structure of this Programming Environment

This programming environment has been developed as a web application to run on a browser. For development, HTML 5, CSS, and JavaScript are used as shown in Figure 4. As a library of JavaScript, we use jquery-sortable [16] which can rearrange nested elements by drag & drop, and SoraMame.Block [17], which is the front end of a lightweight block type code editor.

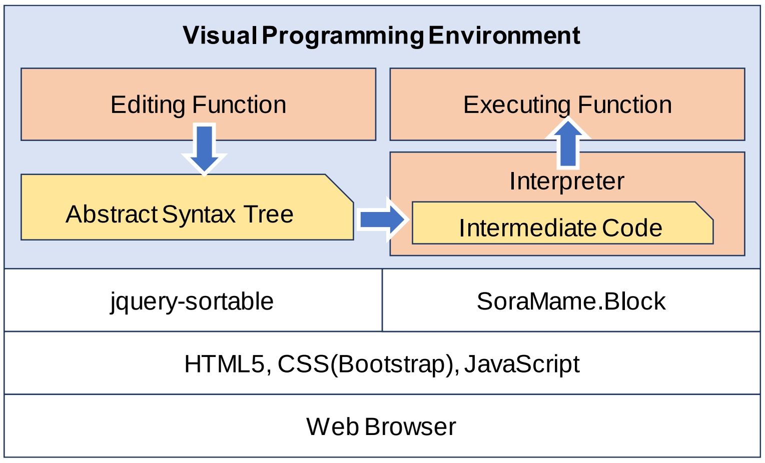

Figure 4: Structure of this Programming Environment

This programming environment has editing and execution functions. The editing function edits the block-based C language source code based on the user's operation, and generates abstract syntax tree (AST) code represented by the proprietary AST language. The executing function executes the AST using the proprietary interpreter based on the user's operation. The details of the AST language and AST are described below in section 4.2, and the details of the interpreter are described in section 4.3.

4.2 AST Language and AST

The editing function generates AST. AST is tree-structured data in which information irrelevant to the meaning of the language is removed from the parse tree.

Each node of AST consists of the type of node and the information necessary for each type. The types of nodes include 'int', 'float', and 'array', representing variable declarations, 'func_dec', representing a function declaration, 'if' and 'while', representing control statements, and 'literal', representing literals. Table 1 shows examples of the types of nodes and the necessary information for each type.

Table 1 AST Language (only a part)

AST node	Item	Description
include statement	Kind	“#include”
	Name	file name
variable declaration	Kind	“int”, “float”, “double”, or “char”
	Name	variable name
	Value	initial value
variable declaration (array)	Kind	“array”
	Type	type of array element
	Name	array name
	Dimension	dimension of array
	Length	length of array
	Value	initial value
function declaration	Kind	“func_dec”
	Type	type of return value
	Name	function name
	Argv	arguments list
	Statements	body of function
if statement	Kind	“if”
	Condition	condition expression
	Statements	list of statement
for statement	kind	“for”
	init	Initialization
	condition	condition expression
	increment	increment expression
	statements	list of statements
function call	Kind	“func_call”
	Name	function name
	argv	Arguments

Figures 5 and 6 show an example of source code and an example of a corresponding abstract syntax tree.

```
# include <stdio.h>

int f ( int n ) {
    return n;
}

int main (  ) {
    printf("f(3) = %d\n", f(3));
}
```

Figure 5: Example of Source Code

```
[
    {
    kind: "#include",  name: "stdio.h"
    },
    {
    kind: "func_dec",  type: "int",
    name: "f",  argv: [{kind: "int", name: "n"}],
      statements: [
         {kind: "return", value: {kind: "var", name: "n"}}
        ]
    },
    {
    kind: "func_dec", type :"int",
    name: "main", argv: [],
        statements: [
              {
                  kind: "printf",
                  format: "f(3) = %d\\n",
                  val_list: [
                      {
                          kind: "func_call",
                          name: "f",
                          argv: [{kind: "literal", type: "int", value: "3"}]
                      }
                  ]
              }
         ]
    }
]
```

Figure 6: Example of AST

```
000 {kind: "#include", name: "stdio.h"}
001 {kind:"_func_dec", name: "f", type: "int", argv: [{kind: "int", name: "n"}],
                                                                    offset: 3}
002 {kind: "return", value: {kind: "var", name: "n"}}
003 {kind: "_func_dec_end", name: "f"}
004 {kind: "_func_dec", name: "main", type: "int", argv: [], offset:3}
005 {kind: "printf", format: "f(3) = %d\\n", val_list: [{kind: "func_call",
                          name: "f", argv: [{kind: "literal", type: "int", value:"3"}]}]}
006 {kind: "_func_dec_end", name: "main"}
007 {kind: "func_call", name: "main", argv: []}
```

Figure 7: Example of Intermediate Code

4.3 Interpreter

In order to enable step execution of a program, the interpreter converts an abstract syntax tree with a tree-structure form into an internal code with a sequential list-structure form, and executes the internal code line by line. This makes it possible to express sentences that are being executed as indices of the list. Also, when converting a tree structure to a list structure, it adds an EOB statement indicating the end of a block, such as an "if" statement. Offset information from the head of the block is added to the EOB statement. Thus, it is possible to obtain the index of the sentence to be executed next. As an example, Figure 7 shows the internal codes of the source code of Figures 5 and 6.

The execution state is called a context. The context has a stack structure as shown in Figure 8, and a function call context or an expression evaluation context is generated for each function call and expression evaluation, respectively, and pushed onto the stack. This stack is called a context stack.

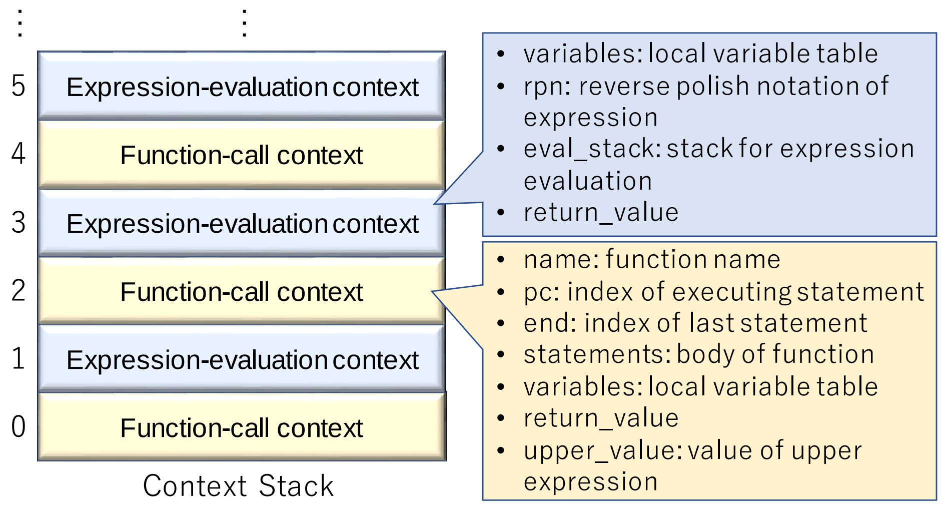
5 Expression-evaluation context
4 Function-call context
3 Expression-evaluation context
2 Function-call context
1 Expression-evaluation context
0 Function-call context
Context Stack
• variables: local variable table
• rpn: reverse polish notation of expression
• eval_stack: stack for expression evaluation
• return_value
• name: function name
• pc: index of executing statement
• end: index of last statement
• statements: body of function
• variables: local variable table
• return_value
• upper_value: value of upper expression

Figure 8: Structure of Context Stack

The context stack is a snapshot of program execution and has all the information necessary for execution. Therefore, by storing the history each time the program executes one step, the context can be reproduced by going back to the steps executed in the past. That is, it is possible to go back 10 steps and re-execute. In this interpreter, the context stacks for 100 steps are saved.

In order to execute the step of evaluating an expression, we convert the expression in the program to Reverse Polish Notation and evaluate it using the stack (called the RPN stack to distinguish it from the context stack). One push of the calculation result to the RPN stack is defined as one step.

Figure 9 shows examples of Reverse Polish Notation for expression '1 + 2 * n2 – n1', 'f(n1, n2)', and 'a[2][1]'. Table 2 shows representation of RPN element for each kind of expression.

```
expression: 1 + 2 * n2 - n1
[
   {"kind":"literal", "value":1, "type":"int"},
   {"kind":"literal", "value":2, "type":"int"},
   {"kind":"var", "name":"n2", "type":"int"},
   {"kind":"*", "argc":2, "type":"int"},
   {"kind":"+", "argc":2, "type":"int"},
   {"kind":"var", "name":"n1", "type":"int"},
   {"kind":"-", "argc":2", "type":"int"}
]

expression: f(n1, n2)
[
   {"kind":"var", "name":"n1", "type":"int"},
   {"kind":"var", "name":"n2", "type":"int"},
   {"kind":"func_call", "name":"f", "argc":2, "type":"int"}
]

expression: a[1][2]
[
    {"kind":"var", "name":"a", "type":"int"},
    {"kind":"literal" ,"value":2, "type":"int"},
    {"kind":"array_elm", "argc":2, "type":"int"},
    {"kind":"literal" ,"value":1, "type":"int"},
    {"kind":"array_elm" ,"argc":2, "type":"int"},
]
```

Figure 9: Example of Reverse Polish Notation (in JSON)

Table 2 Representation of RPN Stack Element

	kind	representation
variable	'var'	kind name type
literal	'literal'	kind value type
binary operator	'+', '-', '*', '/', '%', '&&', '||', '<', '<=', '>', '>=', '==', '!= '	kind argc(=2) type
unari operator	'u+', 'u-', 'u!', 'u*', 'u&'	kind argc(=1) type
assignment	'=', '+=', '-=', '*=', '/='	kind argc(=2) type
prefix/postfix	'pre++', 'pre--', 'post++', 'post--'	kind atgc(=1) type
printf/scanf	'printf', 'scanf'	kind format atgc type
array element	'array_elem'	kind argc(=2) type
function call	'func_call'	kind name argc type

5 Evaluation of This Programming Environment

5.1 Evaluation Method

Some of the above functions have been prototyped, and the effectiveness of this programming environment was evaluated by trial experiments. The scope of supported (and unsupported) C language elements is as follows.

- Type: int, float, double, char, array (struct, union, pointer, enum, typedef, const are not supported)
- Operator: assignment operator, comparison operator, arithmetic operator, logical operator, increment / decrement operator (bitwise operators, shift operators, assignment operators dealing with bits are not supported)
- Control statements: if, if-else, while, for, do-while, break, continue, return (switch is not supported)
- Standard functions: printf, scanf (other standard functions are not supported)
- Syntax error: variable/function not defined
- Run-time error: array index out of range

The subjects were seven undergraduates (five 4th grade students and two 3rd grade students) from the Department of Information Engineering, Kanagawa Institute of Technology. They took a C language course for one year and earned credits. However, they are not proficient in C language. They tackled the programming tasks prepared in advance using both this programming environment and MS Visual Studio. They first used MS Visual Studio to solve four exercises, and then used this tool to solve four similar but different exercises. The outline of the exercise is as shown in Table 3. Coding in this programming environment started with the include statement and main function displayed beforehand in the program pane.

Table 3 Overview of Programming Exercises

	Exercise outline	**Elements of the language to use**	**Number of lines**
1	Display specified character	printf	4
2	Display typed characters	printf, scanf	7
3	Display entered number of specified characters	repetition (for or while)	9
4	After entering the number for the specified number of times, display the value	two repetitions (for or while)	12

In the experiment, the following equipment/software was used.

- PC: Panasonic Let's note CF-SX 4
- OS: Windows 10 Pro
- Development environment: Microsoft Visual Studio Community 2017 (version 15.9.3)
- Browser: Google Chrome (version 71.0.3578.98)

5.2 Evaluation Items

The following evaluation items were compared when coding in this programming environment and coding in Microsoft Visual Studio.

(1) Number of trials until correct answer
(2) Coding time (the time it took from the start of coding to pressing the execution button)

Also, this 5-question survey was taken by those who used this programming environment:

1. Was it easy to use?
2. Was it easy to use the variable block generation function?
3. Were you able to do what you expected?
4. Were you able to understand the movement of the program?
5. Your opinions? (free description)

5.3 Results and Discussion

The results and discussion are described for each evaluation item.

5.3.1 Number of trials until correct answer

Figure 10 shows the number of trials that preceded the correct answer. Each pair of raw scores corresponds to one exercise task described in Section A, the number of subjects correctly responding in one trial (coding, compiling, and execution), the number of subjects correctly responding in two trials, and the number of subjects who required three trials, respectively. The upper row shows the values when this programming environment was used and the lower row shows the values when using MS Visual Studio.

For simple tasks such as outputting and inputting character strings, there was no difference between the programming environment and Visual Studio in the number of trials until the correct answer was produced. However, with a program that used double iteration, coding in this programming environment produced fewer program mistakes. Therefore, for students who are not good at programming, this programming environment is considered to be more effective for complicated programs.

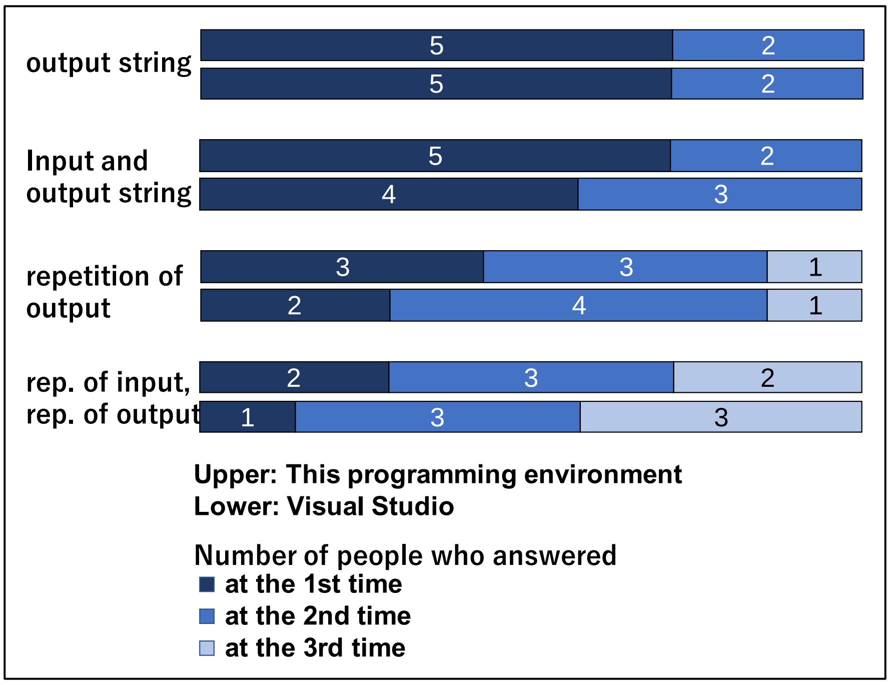

Figure 10: Number of trials until correct answer

5.3.2 Program creation time

The creation time of the program is shown in Figure 11. Each set corresponds to one exercise task described in Section A. Each graph is a box plot showing the time from the start of coding until pressing the execution button. The values on the left side are for the case using this programming environment and those on the right side are for Visual Studio. In simple tasks, there is no difference in program creation time, but as subjects get more complicated, coding using this programming environment is faster. For programmers who are not good at C, it was found that coding in this programming environment is faster if the program is long.

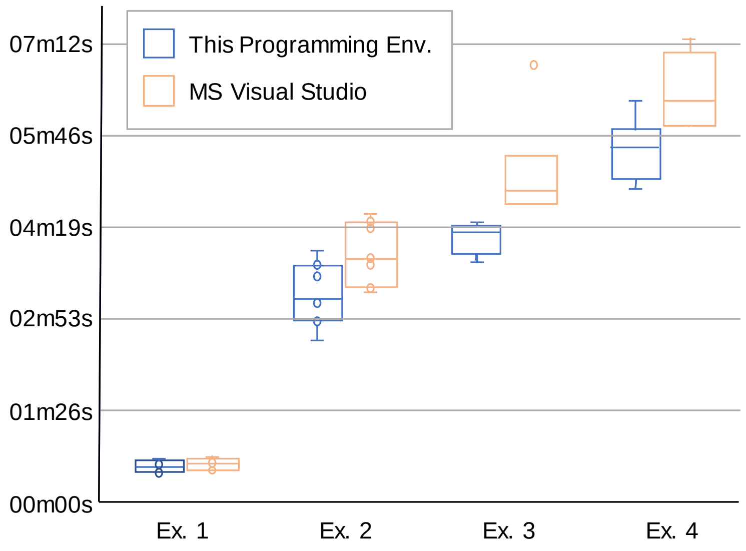

Figure 11: Program creation time

5.3.3 Impressions

The results of a questionnaire on impressions are shown in Figure 12. Each question is then discussed.

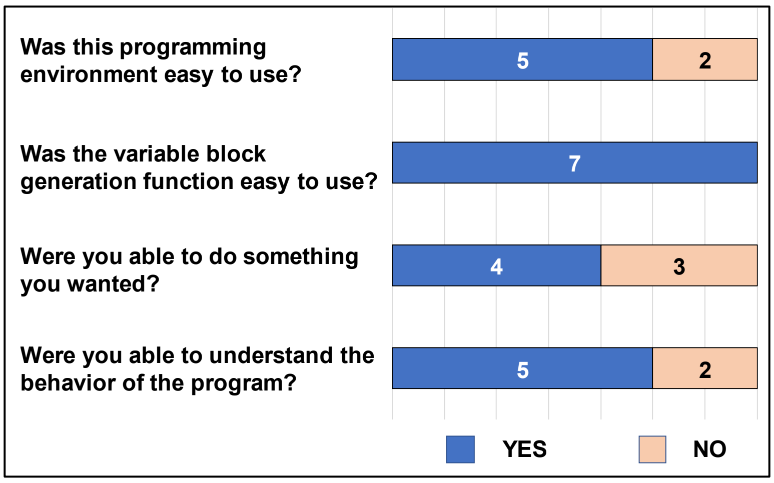

Figure 12: Impressions

Question 1: Was it easy to use?

While there was a positive answer that "I did not have much difficulty because I did not enter any letters," two users said that it was hard to use. The reason given was that "there was a block, but I didn't know how it would work". It is necessary to devise measures such as displaying a use example and explanation of the block when hovering the cursor over the block.

Question 2: Was it easy to use the variable block generation function?

All subjects responded that the variable block generation function was easy to use. This is the effect of reducing input mistakes on variable names by reducing the required labor input for the function.

Question 3: Were you able to do what you expected?

Three answered that they could not do what they thought. This was due to a bug in the variable declaration in the initialization expression of the for statement. These subjects declared a variable like int i = 0; in the initialization expression of the for statement, but this programming environment did not support a variable declaration in the initialization expression, so an error occurred. In this implementation, not all language functions are supported, so there are other syntaxes that do not work. Improvement of the interpreter is necessary.

Question 4: Were you able to understand the action of the program?

Two subjects replied that they could not understand the behavior of the program. There was also an opinion that "I could understand the behavior of the program better if I knew which sentence of the program was being executed". At the time of this experiment, we did not implement the function of step execution, and we believe that the implementation of this function will improve the understandability of behavior.

Question 5: Your opinion is . . .? (free description)

A positive answer was obtained that "I thought it would be good to learn logical thinking" and "I thought that it was easy to understand how to code in a GUI. For a C language novice programmer, it is a good introduction"

6 Conclusion

In order to lower the hurdles for a C language novice programmer, we prototyped a C language visual programming environment. Featured were the following three points: (1) it is possible to edit a program without precisely recalling the C language keywords and syntax by programming with blocks, (2) when declaring variables and functions, the corresponding block can be used. This allows the user to understand the concept of declaration, (3) with the various step execution functions, it is possible to trace the behavior of the program in detail. With the step-back function, it is easy to identify where there was an error.

We evaluated this programming environment by conducting trial experiments and confirmed that students who are not good at C can program more accurately and quickly than by using text-based coding.

Implementation of the functions to visualize the behavior of the program more visually that help understand the programming concepts advanced novices have trouble with [18], improvement of the interpreter, and improvement of usability are future tasks. We will also collect operation logs of this programming environment and develop them into learning analytics.

References

[1] Brett A. Becker, An Effective Approach to Enhancing Compiler Error Messages. Proc. of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16), pp.126-131, 2016.

[2] Brett A. Becker, Kyle Goslin, and Graham Glanville, The Effects of Enhanced Compiler Error Messages on a Syntax Error Debugging Test, Proc. of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18), pp.640–645, 2018.

[3] Scratch - imagine, program, share, https://scratch.mit.edu/

[4] Snap! – build your own blocks, https://snap.berkeley.edu/

[5] D. Bau, J. Gray, C. Kelleher, J. Sheldon, F. Turbak, Learnable programming: blocks and beyond, Communications of the ACM, June 2017, pp. 72-80, 2017.

[6] M. Armoni, O. Meerbaum-Salant, M. Ben-Ari, From Scratch to "real" programming, ACM Transaction on Computing Education, Vol.14, No.4, Article No. 25, 2015.

[7] Y. Matsuzawa, Y. Tanaka, S. Sakai "Measuring an impact of block-based language in introductory programming". In: Brinda T., Mavengere N., Haukijarvi I., Lewin C., Passey D. (eds) Stakeholders and Information Technology in Education. SaITE 2016. IFIP Advances in Information and Communication Technology, vol.493, pp.16-25, Springer, Cham, 2016.

[8] T. W. Price, T. Barnes, Comparing textual and block interfaces in a novice programming environment, Proceedings of the eleventh annual International Conference on International Computing Education Research, pp.91-99, 2015.

[9] Mazyar Seraj, Serge Autexier, Jan Janssen, BEESM, a block-based educational programming tool for end users, Proc. of the 10th Nordic Conference on Human-Computer Interaction, pp.886–891, 2018.

[10] Blockly, https://developers.google.com/blockly/

[11] Y. Matsuzawa, H. Tasui, M. Sugiura, S. Sakai, Seamless language migration in introductory programming education through mutual language translation between visual and Java (in Japanese), Vol.55, No.1, pp.57-71, 2014.

[12] D. Bau, A. Bau, M. Dawson, C. S. Pickens, Pencil Code: block code for a text world, Proceedings of the 14th International Conference on Interaction Design and Children, pp.445-448, 2015.

[13] Davin Mccall, Michael Kolling, A New Look at Novice Programmer Errors, ACM Transactions on Computing Education (TOCE), Article No.: 38, 30pages, 2019.

[14] X. Fu, C. Yin, A. Shimada, H. Ogata, Error log analysis in C programming language courses, Proceedings of the 23rd International Conference on Computers in Education, pp.641-650, 2015.

[15] X. Fu, A. Shimada, H. Ogata, Y. Taniguchi, D. Suehiro, Real-time learning analytics for C programming language courses, Proceedings of the Seventh International Learning Analytics & Knowledge Conference, pp. 280-288, 2017.

[16] jQuery Sortable a flexible, opinionated sorting plugin for jQuery, http://johnny.github.io/jquery-sortable/

[17] MameBlock.js, http://ycatch.github.io/mameblock.js/index.html

[18] Gavriel Yarmish, Danny Kopec, Revisiting novice programmer errors, SIGCSE Bull, Vol.39, No.2, pp.131–137, 2007.