A Study of Phrase Fill-in-Blank Problem for Learning Basic C Programming

Xiqin Lu, Nobuo Funabiki*, Annisa Anggun Puspitasari, and Kiyoshi Ueda

***Abstract*—In a lot of universities across the world, C programming is taught to novice students in the first programming course. To assist their self-studies, we have developed *C programming learning assistant system (CPLAS)* that offers various programming problems with different learning goals where any answer from a student is automatically marked at the system. In this paper, we studied the *phrase fill-in-blank problem (PFP)* in CPLAS for learning basic C programming. A PFP instance gives a source code where several key phrases (set of elements) are blanked and requests to fill in them by a student. The correctness of answers is checked through *string matching* with the original ones. For evaluations, we generated 22 PFP instances for learning *basic grammar topics* and *logic functions* and assigned them with previous 15 instances for *recursive functions* to 21 students in Okayama University. The results confirmed the correctness of the generated instances and found the weakness of students in C programming study.**

***Index Terms*—Introductory C programming, phase fill-in-blank problem, novice student, self-study**

I. Introduction

Currently, in a lot of universities across the world, C programming is taught to novice students in the first programming course, before teaching more advanced and practical programming languages such as Java, Python, and JavaScript. Unfortunately, students are often suffering from studying it due to the formality nature of using symbols and logics programming. To assist their self-studies at homes without teachers, we have developed *C programming learning assistant system (CPLAS)* that offers several types of exercise problems with different leaning goals and levels. In any problem, an answer from a student is marked at the system automatically.

Previously, in CPLAS, we have defined and implemented the *grammar-concept understanding problem (GUP)* [1], the *value trace problem (VTP)* [2], the *element fill-in-blank problem (EFP)* [3], and the *code completion problem (CCP)* [4]. In these problems, one problem instance consists of a source code, a set of questions, and their correct answers. The correctness of any answer is determined through *string matching* with the stored correct answer in the system. The common answer interface for them has been implemented on a web browser [5], where the marking function was implemented by JavaScript that runs on the web browser.

Manuscript received July 8, 2022; revised August 12, 2022; accepted September 19, 2022.

Xiqin Lu and Nobuo Funabiki are with the Department of Electrical and Communication Engineering, Okayama University, Okayama, Japan.

Annisa Anggun Puspitasari is with the Department of Intelligent Mechatronics Engineering, Sejong University, Seoul, South Korea.

Kiyoshi Ueda is with the Department of Computer Science, Nihon University, Koriyama, Japan.

*Correspondence: funabiki@s.okayama-u.ac.jp (N.F.)

In a GUP instance, the source code contains basic keywords in C programming to be studied, such as reserved words and common library elements. Each question describes the grammar concept on a keyword and requests to answer the corresponding element appearing in the source code. It is noted that an element represents the least unit of the source code. GUP aims the *grammar* study.

In a VTP instance, the source code contains several standard output statements for important variables or output messages. Each question requests to answer the corresponding output value. VTP aims the *code reading* study.

In an EFP instance, the source code contains several blank elements with specifying the locations. An element can be a reserved word, an identifier, an operator, and a control symbol here. Each question requests to fill in the blank. EFP aims the *code writing* study.

In a CCP instance, the source code contains several blank elements like EFP, but does not specify the locations. Then, each question requests to find the location of a missing element in the code and fill in it with the proper one. CCP aims the *code reading & writing* study.

For any exercise problem type, every time a student submits the answer for an instance, it is stored in the web browser with the time stamp using *Web Storage* [6]. When the student needs to submit the answers to a teacher, the answers by this student can be saved in a text file to be downloaded from the web browser.

After collecting the answers from the students, the teacher will analyze and tally them by running the *answer analyzer*. This Java program selects only the final answer of the student to each instance using the time stamp. Then, it calculates the average correct answer rate and the average number of submission times for each instance and each student from all the submitted answers by the students. After that, it will output them into the Excel file.

The existing problem types have different learning goals to assist step-by-step programming self-studies of students with CPLAS. Unfortunately, they are not sufficient for the *code writing* study. The question only asks to fill in one element in the source code. To complete a source code, it is necessary to make a combination of multiple elements properly.

Recently, as a new problem type for the *code writing* study of C programming by novice students, we have proposed and studied the phrase *fill-in-blank problem (PFP)* [7]. In a PFP instance, the source code contains several blanks of phrases or sets of multiple elements. Each question requests to fill in the blank with the phrases that originally existed in the code. To generate a new PFP automatically as best as possible, we have implemented the *PFP generator* using Python.

In PFP, one blank can substitute any number of elements in the source code, unlike EFP or CCP where one blank can do

doi: 10.18178/ijiet.2023.13.9.1948

only one element. This flexibility of PFP is able to increase the number of candidates or choices for blanks in the source code, which can enhance learning effects and difficulty levels at solving problems by novice students.

In this paper, we studied the *phrase fill-in-blank problem (PFP)* in CPLAS for learning basic C programming. Collecting proper source codes from [8–12] and running the *PFP generator*, we newly generated 13 PFP instances for learning *basic grammar topics* and nine instances for learning *logic functions*. Then, we assigned them with previously generated 15 instances for *recursive functions*, to 21 students in Okayama University.

For evaluations of the PFP instances, we collected the answer files from the students, and calculated the average correct rate and the number of submission times from them using the *answer analyzer*. Then, from them, we could confirm the correctness of the generated PFP instances and find the weakness of the students in C programming topics and the students to be cared.

The rest of this paper is organized as follows: Section II provides a review of related works in literature. Section III presents the PFP instances generated for this application. Section IV shows the example of PFP instance. Section V shows their application results to novice students and discussions. Finally, Section VI concludes this paper with future work.

II. Related Works

In this section, we discuss related works in literature.

In [13], Freund *et al*. developed the *Thetis* programming environment that is designed specifically for student to use. This system consists of the C interpreter and the associated user interface to provide simple and easily-understood editing, debugging, and visualization capabilities. It is more suitable for students in introductory computer science courses, because some commercially available compilers, particularly those used for languages like ANSI C, assume the level of sophistication that novice students do not possess.

In [14], Boada *et al*. presented a web-based tool, named *ACMEp*, with the aim of reinforcing teaching and learning of introductory programming. From the teacher's perspective, this system can introduce important gains with respect to the classical teaching methodology where the teacher can perform continuous assessments of progresses of students. From the student's perspective, it provides a learning framework for programming and correction environments with helps, which facilitates their personal works.

In [15], Bravo *et al*. described several educational computer tools, named *SICAS*, *PlanEdit*, *COLLEGE*, and *OOP-Anim*, to support programming learning, and presented a global environment of integrating them, allowing a broader approach to programming teaching and learning. This environment offers animations and the *computer-supported collaborative learning (CSCL)* paradigm.

In [16], Li *et al*. presented a game-based learning environment to support programming learning. It exploits game construction tasks to make the elementary programming more intuitive to learn, and comprises concept visualization techniques, to allow students to learn key concepts in programming through game object manipulation.

III. Phrase Fill-in-Blank Problem for Basic C Programming

In this section, we present the design goal and the generation of the *phrase fill-in-blank problem (PFP)* for learning basic C programming.

A. Design Goal of PFP

For the introductory *coding* study, EFP performs the limited programming activity of filling in only one element to each blank in the given source code. As the next step *coding* study, PFP is designed to work on the more realistic programming activity of filling in multiple elements together to one blank in the code. The number of elements in one blank has no limitation, as long as the blank elements belong to one statement in the code and can give the unique answer to the blank. These constraints come from *string matching* at the marking function.

B. PFP with Blank Phrases

For learning basic C programming efficiently, first, novice students should learn how to write *standard input/output functions* in the source code because they are necessary for visualizing the behaviors of the source code. Second, they should learn how to use the *conditional statements* using `if` for the control flow. Third, they should learn how to give the input and output of a function. By knowing the proper input/output of a function, it is expected that the whole source code can be understood. Fourth, they should learn the *conditional statement* for looping using `while` and `for`. Last, they should learn r*ecursive functions* and other important concepts in C programming.

In this paper, we generated 13 PFP instances for *basic grammar concepts* and nine instances for *logic functions*.

As summary, the following phrases or sets of elements are blanked from the given C source codes for *basic grammar concepts*:

1) the conditional statement of `if` and `while`
2) the statement of `printf` and `scanf`.

Then, the following phrases or sets of elements are additionally blanked from the source codes for *logic functions*, where 5) is added to allow a teacher to make blanks flexibly:

1) the conditional statement of `for`,
2) the argument of the `function`,
3) the returning phrase of `return`,
4) the `header file`,
5) the phrases selected by the teacher manually.

C. Alternative Answer

Unfortunately, the correct answer for a blank may not be unique. Besides the original phrase removed in the source code, other correct phrases may exist. They are called *alternative answers* in the paper, where the alternative answers are found and generated manually.

In a PFP instance, the blanked source code is shown to students who are requested to be filled in with the proper phases. The correctness of each student answer is marked

through *string matching* with the original phase in the source code or the alternative answer if registered.

D. PFP Level

To make step-by-step studies through solving PFP instances at different levels, a teacher can select one of the following four levels at PFP instance generations:

- Level 1: argument of printf, argument of scanf,
- Level 2: (+) condition of if,
- Level 3: (+) condition of while, argument of function, returning value of function,
- Level 4: (+) condition of for, common library, other concept selected by teacher.

"(+)" indicates that the phrases in the lower levels are included at that level. Level 1 considers only the standard input and output. Level 2 considers the if condition in addition to Level 1. Level 3 considers the while condition, and the argument and the returning value of the function, in addition to Level 2. Level 4 considers the for loop condition and the common library, in addition to Level 3. Here, a teacher can choose any phrase in the source code if necessary.

E. Code Specification

In a PFP instance, the code specification is given together when it is necessary to reach the unique correct answer. For example, to answer the argument of printf, the actual standard output message is necessary. Also, to answer the conditional statement of while, the corresponding behavior of the code should be described there.

F. Generation of PFP Instance

A new PFP instance can be generated by the following procedure:

1) To select a proper source code that includes the programming concepts to be studied by this instance,
2) To select one PFP level,
3) To run the PFP generator which is implemented by python, to automatically find and remove the corresponding phrases in the code to this level for the blanked source code, keep the removed phrases for the correct answers, add the alternative answer phrases for the correct answers if necessary, and combine them into one text file,
4) To run the answer interface generator in [5] with this text file, to generate the CSS/HTML/JavaScript files for the answer interface on the web browser,
5) To add the code specifications into the HTML file, if necessary to reach the correct answers,
6) To register the generated PFP instance as an assignment to the students.

IV. Example PFP Instance

In this section, we discuss the example PFP instance where Level 3 is selected.

A. Example Source Code

BOX1 shows the example source code that accepts four positive integers from the standard input and outputs the maximum value among them. This code is used for the first PFP instance (ID=1) for learning *basic grammar concepts.*

BOX1: C source code for *Maximum*

```
#include <stdio.h>
int main(void) {
   int i, max, value;
   for (i = 1; i <= 4; i++) {
      printf("Value No.%d:", i);
      scanf("%d", &value);
      while (value <= 0) {
            printf("Value is not a positive number\n");
            printf("Value No.%d:", i);
            scanf("%d", &value);
      }
      if (value > max)
            max = value;
   }
   printf("The maximum value is %d\n", max);
   return 0;
   }
```

B. Text File to Answer Interface Generator

BOX2 shows the corresponding input text file to the answer interface generator for this PFP instance. The conditional statements of if and while, and the statements of printf and scanf are blanked. The correct answers are stored in the system when they are blanked from the source code automatically. The correct answer for each blank is separated by ",,". Since the system can register up to two correct answers for each blank to consider alternative answers, the two answers are separated by "- - -" if exist. In this instance, value>max and max<value are registered as the correct answers for (_7_).

BOX2: Text input file for generator

```
#include <stdio.h>
int main(void) {
   int i, max, value;
   for (i = 1; i <= 4; i++) {
      printf( _1_ );
      scanf( _2_ );
      while ( _3_ ) {
            printf( _4_ );
            printf( _5_ );
            scanf( _6_ );
      }
      if ( _7_ )
            max = value;
   }
   printf( _8_ );
   return 0;
}

"ValueNo.%d:",i,,"%d",&value,,value<=0,,"Valueisnotaposi
tivenumber\n",,"ValueNo.%d:",i,,"%d",&value,,value>max-
--max<value,," Themaximumvalueis%d\n",max,,
```

C. Code Specification

BOX3 shows the code specification for this PFP instance. To describe the output messages of the code, it includes the

standard input/output example when the code is executed.

BOX3: Code specification

Create a program that inputs four positive integer values and outputs the largest value. However, if a negative value is entered, re-enter it. An example of output: Value No.1:20 Value No.2:-10 Value is not a positive number Value No.2:0 Value is not a positive number Value No.2:10 Value No.3:100 Value No.4:50 The maximum value is 100

D. Answer Interface

Fig. 1 shows the answer interface to this example PFP instance. The left side shows the source code with the input forms to be filled in. When a student clicks the "Answer" button, the correctness of the input answers is checked through string matching, and the background of the incorrect answer becomes red while the correct one keeps white. In this figure, the incorrect answer should be corrected to `value<=0`. The right side shows the code specification, which is given to solve this PFP instances.

E. Data Analysis Program

After a student completes answering the given PFP instances, he/she will submit the answer text file. BOX4 shows the example. The first column in the file represents the student ID number. The second column does the PFP instance ID. The third and fourth columns do the submission date and time. The last column does the submitted answers and their making results, where those for each question are separated by ",". `[o]` indicates the correct answer and `[x]` does the wrong one respectively. Since any submission record is stored in the text file, it becomes difficult for a student to copy the file of other student and submit it to the teacher.

After collecting the answers from the students by emails or a file server, the teacher will calculate the average correct answer rate and the average number of submission times by running the *answer analyzer* and analyze them using the output Excel file.

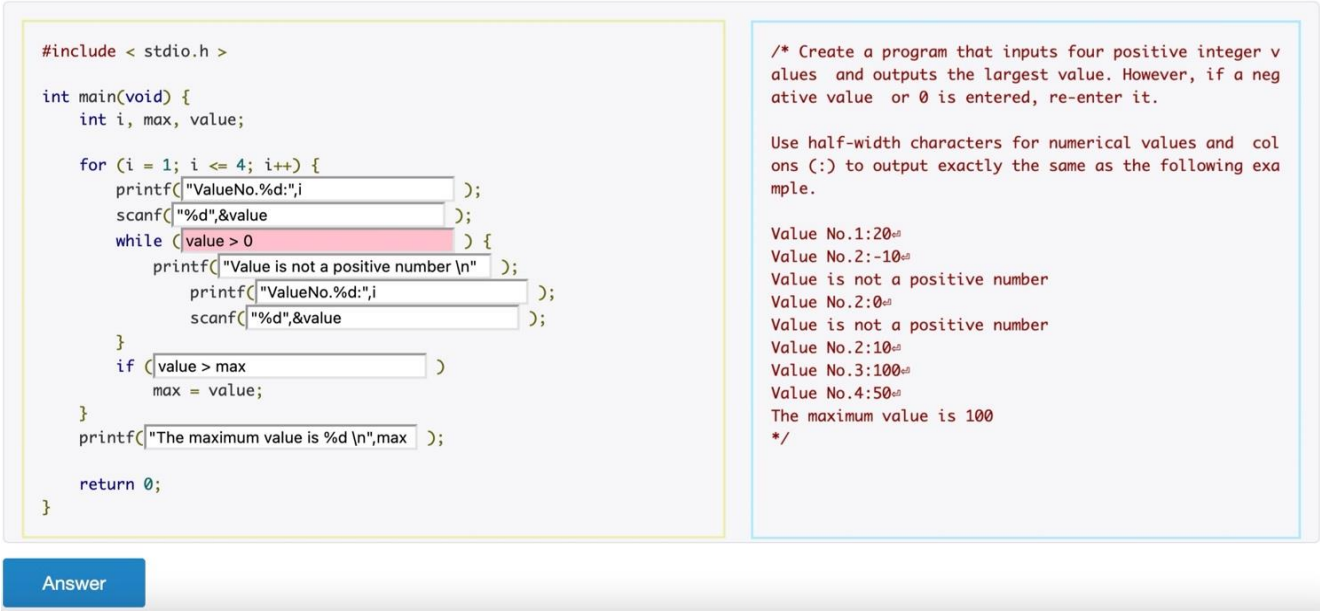

Fig. 1. PFP answer interface.

BOX4: Answer text file

123	3	2022-06-14	18:58:25	n1[o],num!=0[o],sum+=ld*ld*ld[o],num==n1[o]
123	9	2022-06-14	19:18:01	struct course *[o],[x],struct course[o],ptr[o]
123	8	2022-06-14	19:11:24	int*[o],sizeof(int)*n[o],[x],ptr[o]
123	1	2022-06-14	18:52:18	arr1[o],i<n[o],mxelem<arr1[i][o],i++[o],mxelem[o]
123	2	2022-06-14	18:53:08	&n1,&n2[o],tmp=*p[o],*p=*q[o],*q=tmp[o]
123	4	2022-06-14	19:00:23	1[o],2[o],3[o],4[o],5[o],num<n[o],f*=num+1[o],f[o]
123	7	2022-06-14	19:08:34	arr_size<=1[o],arr_size[o],i<arr_size[o],nums[i]==nums[j][x],count[o],nums,size[o]
123	7	2022-06-14	19:06:55	arr_size<=1[o],[x],i<arr_size[o],[x],count[o],nums,size[o]
123	9	2022-06-14	19:13:08	int*[x],[x],noOfRecords[x],ptr[o]
123	5	2022-06-14	19:01:50	n1[o],num%i==0[o],n1==sum[o]
123	9	2022-06-14	19:19:17	struct course *[o],noOfRecords[o],struct course[o],ptr[o]
123	7	2022-06-14	19:07:18	arr_size<=1[o],arr_size[o],i<arr_size[o],[x],count[o],nums,size[o]
123	6	2022-06-14	19:04:25	stLimit, enLimit[o],n1%i==0[o],sum==n1[o],1[o],0[o],stLimit<=enLimit[o],stLimit[o]
123	4	2022-06-14	19:00:17	1[o],2[o],3[o],4[o],5[o],num<n[o],f*=num[x],f[o]

```
123  9  2022-06-14  19:15:04  course[x],[x],[x],ptr[o]
123  1  2022-06-14  18:52:06  arr1[o],i<n[o],mxelem<arr1[x],i++[o],mxelem[o]
123  9  2022-06-14  19:15:45  struct course *ptr[x],struct course[x],struct course[o],ptr[o]
123  9  2022-06-14  19:19:08  struct course *[o],ptr[x],struct course[o],ptr[o]
```

V. Evaluation

In this section, we discuss the evaluation of the *phrase fill-in-blank problem (PFP)* for learning basic C programming. We generated PFP instances at Level 3 or Level 4 for learning *basic grammar topics*, *logic functions* and *recursive functions*, and assigned them to undergraduate students in Okayama University, Japan.

A. Evaluation for Basic Grammar Topics

First, we discuss the evaluation of PFP instances for basic grammar topics.

1) Generated PFP instances

Table I shows the instance ID, the program topic, the number of lines in the source code (LOC), and the number of answer forms in each of the 13 PFP instances.

TABLE I: PFP Instances for Basic Grammar Topics

ID	topic	# of lines	# of forms
1	Maximum	17	8
2	Multiplication table	13	3
3	Odd and even number	13	5
4	Large small judgment	19	12
5	Standard output	10	2
6	Triangle area	19	5
7	Total	11	2
8	Error message	14	5
9	Calculation	12	4
10	Output of array	13	4
11	Sum of array	13	2
12	Value of array	14	4
13	Struct	12	2

2) Solution results of individual instances

Fig. 2 shows the average number of answer submission times and the average correct answer rate (%) by all the 21 students of each of the 13 PFP instances. The instance at ID=10 gave the lowest correct rate and highest submission time, which suggests that *array* was the hardest basic grammar topic for beginners. The instance at ID=13 showed the highest correct answer rate, because this instance asked only the printf statement.

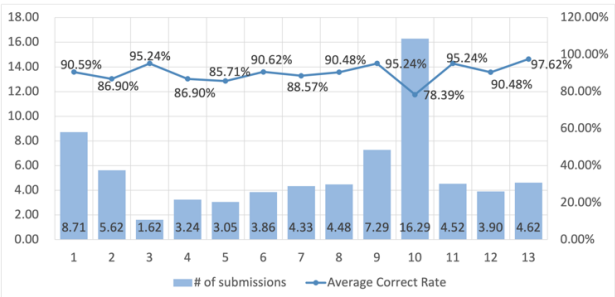

Fig. 2. Solution results of individual instances for basic grammar topics.

3) Solution results of individual students

Fig. 3 shows the average number of answer submission times and the average correct answer rate among the 13 instances of each student. The 11 students at ID=2, 3, 4, 6, 8, 10, 11, 17, 18, 19 and 21 achieved the 100% correct rate for any instance. The student at ID=14 took the highest submission times, which means this student solved them seriously but needs improvements in C programming. The two students at ID=12 and 15 showed the low correct rates 43.94% and 18.18%. They need more efforts in studying basic grammar topics of C programming, which should be watched and cared by the teacher.

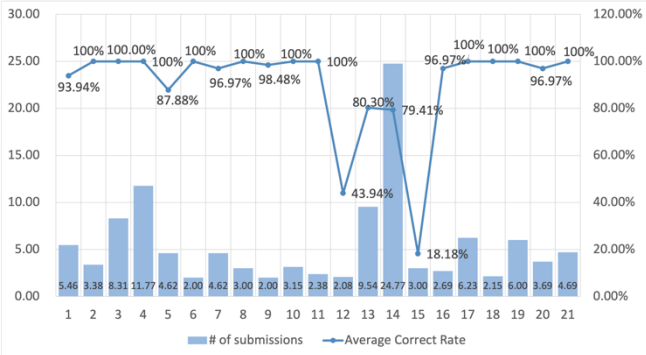

Fig. 3. Solution results of individual students for basic grammar topics.

B. Evaluation for Logic Functions

Next, we discuss the evaluation of PFP instances for *logic functions.*

1) Generated PFP instances

Table II shows the instance ID, the logic function topic, the number of lines in the source code, and the number of answer forms for each of the nine PFP instances.

TABLE II: PFP Instances for Logic Functions

ID	Topic	# of lines	# of forms
14	Largest element	27	5
15	Swap numbers	19	4
16	Armstrong number	23	4
17	Sum of series	17	8
18	Check perfect number	23	3
19	Print perfect numbers	33	7
20	Remove duplicates	32	6
21	Calculate sum	21	4
22	Allocate memory	23	4

2) Solution results of individual instances

Fig. 4 shows the average number of answer submission times and the average correct answer rate (%) by all the students for each instance. The instance at ID=22 results in the lowest correct rate. *Memory allocation* will be the difficult function for them, where it needs to understand the malloc

and free functions. The instance at ID=20 took the highest submission times, where it has several blanks for ***if*** and ***while*** statements in the recursive function.

3) Solution results of individual students

Fig. 5 shows the average number of answer submission times and the average correct answer rate among the nine instances for each student. The five students at ID=1, 3, 4, 18, and 19 achieved 100% correct rate. Again, the same students at ID=12 and 15 showed the low correct rates and the low submission times.

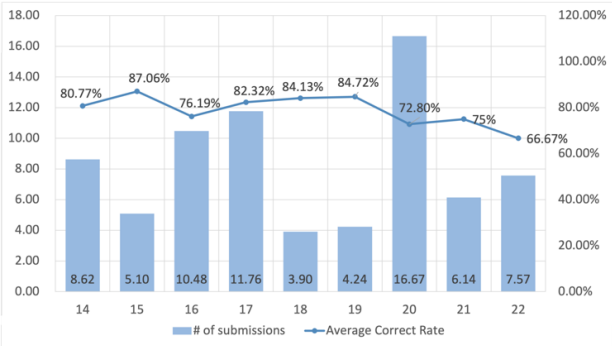

Fig. 4. Solution results of individual instances for *logic functions.*

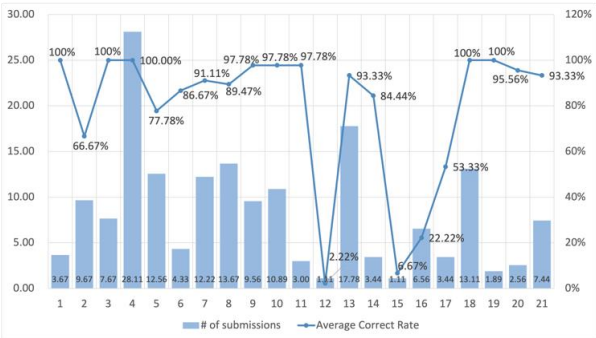

Fig. 5. Solution results of individual students for *logic functions*.

C. Evaluation for Recursive Function

Finally, we review the evaluation of the PFP instances for *recursive functions* in [7].

1) Generated PFP instances

Table III shows the instance ID, the recursive function topic, the number of lines in the source code, and the number of answer forms for each instance.

TABLE III: PFP INSTANCES FOR RECURSIVE FUNCTIONS

ID	topic	# of lines	# of forms
23	Power function	15	4
24	Sum of digits	18	5
25	Count digits	19	5
26	Factorial of number	15	4
27	Fibonacci series	22	5
28	String length	19	4
29	Sum of numbers	19	5
30	Greatest common divisor(GCD)	19	4
31	Reverse string	19	4
32	Decimal to binary	20	5
33	Prime number	26	7
34	Even and odd numbers	18	4
35	Tower of Hanoi	13	3
36	Levenshtein distance	13	6
37	Ackermann function	34	9

2) Solution results of individual instances

Fig. 6 shows the average number of answer submission times and the average correct answer rate (%) by all the students for each of the 15 instances. The instance at ID=35 exhibited the lowest correct rate. *Tower of Hanoi* can be difficult for the students because it recursively calls the same function twice with the different arguments. The instance at ID=37 showed the highest submission times, where they may be not familiar to the source code of the *Ackermann Function* using the *recursive function*.

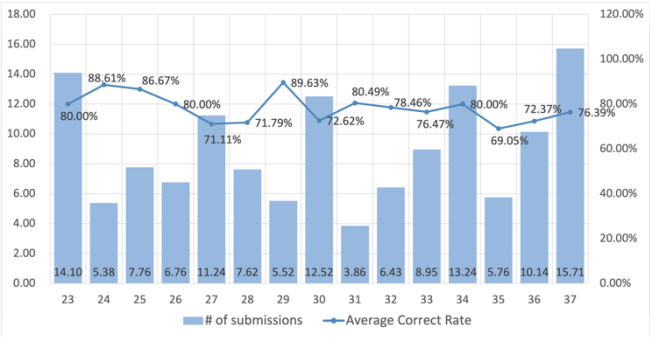

Fig. 6. Solution results of individual instances for *recursive functions*.

3) Solution results of individual instances

Fig. 7 shows the average number of answer submission times and the average correct answer rate among the 15 instances by each student. The four students at ID=3, 4, 18, and 19 achieved the 100% correct rate, and the seven students at ID=5, 8, 12, 14, 15, and 17 did under 80%.

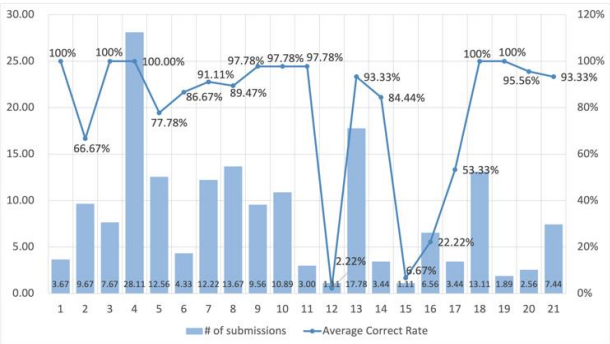

Fig. 7. Solution results of individual students for *recursive functions.*

D. Discussions

Here, we discuss the following research questions on the effectiveness of PFP for learning basic C programming:

1) The generated PFP instances can be correctly marked with manually added alternative answers?
2) Most of C programming novice students can solve PFP instances by themselves?
3) The teacher can find the students who need cares in the C programming course?

For the first question, the answer is yes, because some students solved all the questions correctly. For the second question, the answer can be yes or no, because the correct rates in many instances exceed 80% whereas in some PFP instances, they were lower than 70%. It will be necessary to give hints for hard questions that cannot be solved even after several submissions, which will be in future works. For the last question, the answer is yes, because the two students at

ID=12 and 15 showed the very low correct rates and submission times.

VI. Conclusion

This paper studied the phrase *fill-in-blank problem (PFP)* for learning basic C programming. 13 instances for *basic grammar topics*, nine instances for *logic functions*, and 15 instances for *recursive functions* were generated using C source codes containing the corresponding concepts. The application results of the 37 PFP instances to 21 undergraduate students confirmed the correctness of the generated instances and found the weakness of students in C programming study. In future works, we will implement hint functions for hard PFP instances, generate new PFP instances for other topics in C programming, and evaluate them through applications to novice students.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

All the authors conducted the research together. Particularly, Xiqin Lu generated the problems, analyzed the result data, and wrote the paper. Nobuo Funabiki supervised the activities and refined the paper. Kiyoshi Ueda assigned the generated problems to students and Annisa Anggun Puspitasari collected the data. All the authors had approved the final version.

References

[1] S. T. Aung, N. Funabiki, Y. W. Syaifudin, H. H. S. Kyaw, S. L. Aung, N. K. Dim, and W.-C. Kao, "A proposal of grammar-concept understanding problem in Java programming learning assistant system," *J. Adv. Inform. Tech*., vol. 12, no. 4, pp. 342–350, Nov. 2021.

[2] X. Lu, N. Funabiki, H. H. S. Kyaw, E. E. Htet, S. L. Aung, and N. K. Dim, "Value trace problems for code reading study in C programming," *Adv. Sci. Tech. Eng. Syst. J. (ASTESJ)*, vol. 7, no. 1, pp. 14–26, Jan. 2022.

[3] N. Funabiki, Tana, K. K. Zaw, N. Ishihara, and W.-C. Kao, "A graph-based blank element selection algorithm for fill-in-blank problems in Java programming learning assistant system," *IAENG Int. J. Comput. Sci*., vol. 44, no. 2, pp. 247–260, 2017.

[4] H. H. S. Kyaw, S. T. Aung, H. A. Thant, and N. Funabiki, "A proposal of code completion problem for Java programming learning assistant system," in *Proc. VENOA-2018*, pp. 855–864, July 2018.

[5] N. Funabiki, H. Masaoka, N. Ishihara, I.-W. Lai, and W.-C. Kao, "Offline answering function for fill-in-blank problems in Java programming learning assistant system," in *Proc. ICCE-TW*, pp. 324-325, May 2016.

[6] P. Lubbers, B. Alers, and F. Salim, *Pro HTML5 Programming*, 2010, pp. 213–241.

[7] X. Lu, S. Chen, N. Funabiki, M. Kuribayashi, and K. Ueda, "A proposal of phrase fill-in-blank problem for learning recursive function in C programming," in *Proc. LifeTech*, pp. 127–128, Mar. 2022.

[8] W3resourceurl: Function, array. [Online]. Available: https://www.w3resource.com/c-programming-exercises/function/index.php

[9] Programiz: Store data in structures dynamically. [Online]. Available: https://www.programiz.com/c-programming/examples/structure-dynamic-memory-allocation

[10] J. H. Conway and R. K. Guy, *Perfect Numbers*, New York: Springer-Verlag, pp. 136–137, 1996.

[11] Recursion in C. [Online]. Available: https://www.javatpoint.com/recursion-in-c

[12] C programming recursion examples, programs. [Online]. Available: https://www.includehelpcom/c-programs/recursion-examples.aspx

[13] S. N. Freund and E. S. Roberts, "Thetis: An ANSI C programming environment designed for introductory use," *SIGCSE Bull*., vol. 28, no. 1, pp. 300–304, Mar. 1996.

[14] I. Boada, J. Soler, F. Prados, and J. Poch, "A teaching/learning support tool for introductory programming courses," in *Proc. ITHET*, pp. 604–609, May 2004.

[15] C. Bravo, M. J. Marcelino, A. Gomes, M. Esteves, and A. J. Mendes, "Integrating educational tools for collaborative computer programming learning," *J. Univers. Comp. Sci*., vol. 11, no. 9, pp. 1505–1517, Feb. 2005.

[16] F. W. B. Li and C. Watson, "Game-based concept visualization for learning programming," in *Proc. MTDL*, pp. 37–42, Dec. 2011.

Copyright © 2023 by the authors. This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited (CC BY 4.0).