CB-003

Applications of Element Fill-in-blank Problems for C Programming Learning in Three Universities

Htoo Htoo Sandi Kyaw‡ Nobuo Funabiki‡ Thandar Myint‡‡ Phyu Phyu Tar‡‡
Nandar Win Min‡‡ Hnin Aye Thant‡‡ Shune Lae Aung‡‡‡ Nem Khan Dim‡‡‡

pxs93q36@s.okayama-u.ac.jp funabiki@okayama-u.ac.jp

Abstract Presently, C programming is essential for IT students to study programming languages, algorithms, and computer architecture. By extending our works for *Java programming learning assistant system (JPLAS)*, we are currently developing *C programming learning assistant system (CPLAS)* for self-studies by students. CPLAS offers the *element fill-in-blank problem (EFP)* that asks students to fill in the blank elements in the given source code. The correctness of any answer is marked through *string matching*. In this paper, we investigate the effectiveness of EFP in CPLAS at *C programming* educations by analyzing the application results in three universities. We generate 19 EFP instances using C source codes on 1) basic grammar concepts and 2) fundamental data structures & algorithms and assign them to undergraduate students in one university in Japan and two universities in Myanmar. The results observed that the average correct answer rate increases as the number of answer submissions increases for 1), the rate is highest at one university where the submission number is lowest for 2), and one university gives the lowest rate for both. The teachers should feedback these observations at their C programming educations.

1. Introduction

Nowadays, *C programming* has still been considered to be the most fundamental programming language. A lot of universities around the world provide C and C++ programming language courses, not only in IT departments but also in other departments such as mechanical engineering and electrical engineering. Besides, students in IT departments should study C programming in parallel with computer architecture courses, because it needs, for example, to study the access to memories or registers for efficient programming on the given computer architecture. As a result, C programming language is selected as the third most popular programming language despite the age since the appearance [1].

Previously, to assist self-study of *Java programming,* we have developed a Web-based *Java Programming Learning Assistant System (JPLAS)* [2]. Java has been widely used in IT societies as the practical object-oriented programming language. Thus, Java programming has been educated in many IT departments. JPLAS provides several types of exercise problems to cover different study levels of Java programming studies. Then, for any exercise problem, the answer from a student is marked automatically using the software in JPLAS, to support self-studies of Java programming.

For the first learning stage, JPLAS offers the *element fill-in-blank problem (EFP)* [3]. The EFP is designed for novice students to study Java grammar and basic programming skills through *code reading.* In the EFP instance, a Java source code that has several blank elements is given to students, where the blanks are shown explicitly in the code. Then, the students are requested to fill in the blanks by typing the correct elements. Here, an *element* represents the least unit in the code, which includes a *reserved word*, an *identifier*, and a *control symbol*. The correctness of the answer from a student is verified through *string matching* with the corresponding original element in the code.

In the EFP instance, the original element in the source code for any blank must be the unique correct answer, to avoid confusions among novice students. Thus, we proposed the graph-based *blank element selection algorithm* to select blank elements automatically. This algorithm can select as many blanks as possible that have grammatically correct and unique answer elements in the source code. The EFP in JPLAS has successfully been applied to students in universities in Japan, Myanmar, and Indonesia, where a lot of EFP instances have been generated to be solved.

To assist self-study of *C programming*, we are currently developing *C Programming Learning Assistant System (CPLAS)* by extending our works on JPLAS [4]. Then, to automatically generate an EFP instance for C, the *blank element selection algorithm* is newly designed and implemented for C programming where the conditions for selecting blank elements are redefined [4]. This algorithm consists of the four steps: 1) it analyzes the C source code using the original *parser* , 2) it generates a *constraint graph* by selecting each candidate blank element in the code as a *vertex* and connecting any pair of vertices by an *edge* such that their incident elements cannot be blanked simultaneously for unique correct answers, 3) it derives the *compatibility graph* by taking the complement of the constraint graph, and 4) it seeks a *maximal clique* of the compatibility graph to find a maximal set of blank elements with unique answers.

In this paper, we investigate the effectiveness of EFP in CPLAS at *C programming* educations by analyzing the application results in three universities. We generate 1) nine EFP instances using C source codes on basic C grammar concepts and 2) ten EFP instances on fundamental data structures & algorithms, Then, we ask undergraduate students in one university in Japan and two universities in Myanmar to solve them using the answer interface on a browser in [5].

From the results, we observe that the average correct answer rate increases as the number of answer submissions increases for 1), the rate is highest at one university where the submission number is lowest for 2), and one university gives the lowest rate

† Graduate School of Natural Science and Technology
Okayama University, Japan
‡‡Department of Information Science and Technology
University of Technology (Yatanarpon Cyber City), Myanmar
‡‡‡Department of Computer Studies
University of Yangon, Myanmar

Copyright © 2021 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

for both. It is expected that the teachers in these universities should feedback these observations at their *C programming* educations.

The rest of this paper is organized as follows: Section II introduces related works. Sections III reviews the element fill-in-blank problem in CPLAS. Section IV shows the applications of EFP and investigate the effectiveness at C programming educations. Finally, Section V concludes this paper with future work.

2. Related Works

A lot of works has been reported for *C programming* educations. In this section, we introduce related works to the fill-in-blank selection algorithm.

In [6], Kashihara et al. proposed a method of blanking an important point of data or control flow in a C code using *Program Dependence Graph (PDG),* to make instructive fill-in-blank problems without considering semantic aspects. PDG can represent the relationship of the data dependency and the control flows between commands using a graph. By solving the problem, students can improve skills of processing the flow of the program.

In [7], Shinkai et al. provided a C programming education assistant system on *Moodle* using fill-in-blank problems like in this paper. It extracts important elements in a code for questions using PDG.

In [8], Terada et al. proposed a methodology to automatically generate fill-in-blank problems for C codes. To automatically generate problems, two key constituents, the selection of exemplary code and the selection of places to be blanked, are presented. For the first one, they use k-means clustering with silhouette analysis is used to select exemplary code from the *Aizu Online Judge system (AOJ)* which has over three million of source codes. For the later one, a model based on a *bidirectional Long Short-Term Memory Network (Bi-LSTM)* with a sequential *Conditional Random Field (CRF)* is used. The blank selection methodology selects the important places in the process flow of the source code and creates blanks there. In the future, we will consider the use of process flow of the source code in our study.

In [9], Parsons et al. proposed *Parson 's Programming Puzzles*, as the automated, interactive tool. It provides practices with basic programming principles in an entertaining puzzlelike format. The careful design of the items in the puzzles allows the tutor to highlight particular topics and common programming errors. Because each puzzle solution is a complete sample of a well written code, the use of this tool exposes students to good programming practices.

In [10], Brusilovsky et al. developed the *QuizPACK* system that can generate parameterized exercises for the C language and automatically evaluate the correctness of student answers by comparing them to the correct ones provided by the teacher. The fill-in-blank in *QuizPACK* asks the values of particular variables in a source code and is similar to the *value trace problem* in [11][12]. On the other hand, the fill-in-blank problem in this paper requests filling in the blank elements that are composed of reserved words, identifiers, or control symbols for basic grammar and code reading studies.

In [13], Kakeshita et al. developed a programming education support tool called *Pgtracer*. *Pgtracer* utilizes fill-in-the-blank questions composed of a source code and a trace table. The blanks in the code and the trace table must be filled by the students to improve the code reading while solving the questions. In Pgtracer , they are manually selected by the teacher. On the other hand, the blanks are automatically selected using the blank element selection algorithm in our proposal.

In [14], Jain et al. developed an educational tool for understanding algorithm building and learning programming language. This tool provides an innovative and a unified graphical user interface for developments of multimedia objects, educational games, and applications. It also provides an innovative method for code generations to enable students to learn the basics of programming languages using drag-n-drop methods for image objects.

In [15], Cavus et al. conducted an experimental study of a highly interactive and collaborative virtual teaching environment by supporting *Moodle* LMS with collaborative learning tool, *GREWP* tool. The aim of this study is to find out the success rate of students between the teaching using an advanced and a standard collaborative tool and the traditional teaching. The results show that a higher success rate is achieved when the LMS system is combined with the advanced collaborative tool.

In [16], Barros et al. developed an e-learning tool called *ProPAT*. This tool is implemented as an *Eclipse plug-in* with two perspectives: the *teacher perspective* and the *student perspective.* This tool allows students in the first computer science course to learn how to program using *pedagogical patterns*, which are the set of programming patterns recommended by computer science educators. For answer marking, the tool includes the program diagnosis system that uses *Model Based Diagnosis* techniques. The difficulty of the tool is that a teacher needs to collect the *programming patterns*. In our proposal, a teacher only needs to select source codes.

3. Review of Element Fill-in-blank Problem in CPLAS

In this section, we review our works on EFP in CPLAS in [4].

3.1 Element Definition in EFP for C

An *element* in EFP for C programming is defined as the least unit of a source code. In an EFP instance, a reserved word, an identifier, a conditional operator, a memory operator, a preprocessor, and a control symbol can be blanked among the elements in the code, if it gives the unique answer.

A *reserved word* signifies a fixed sequence of characters that has been defined in C grammar to represent a specific function. It is expected that students should master the proper use in learning programming. An *identifier* is a sequence of characters defined in the code by the author to represent a *variable* or a *function*. A *conditional operator* is used in a conditional statement to determine the state. A *memory operator* is used as the pointer to a variable * or to get address of the variable &. A *preprocessor* includes #, include, define, <, >, . , and h for a *header file* . A *control symbol* in the paper indicates other grammar element, including . , : , ; , ( ) , and { } .

Copyright © 2021 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

3.2 Example EFP Instance

code 1 shows an example EFP instance. This instance is generated from **code 2**. On the blank elements, # at _1_ is the preprocessor directive, . at _2_ and h at _3_ are for the header file extension, main at _4_ is the identifier, int at _5_ is the reserved word for data type, ; at _6_ is the control symbol, printf at _7_ is the identifier, ; at _8_ is the control symbol, & at _9_ is the memory operator, number at _10_ is the identifier, and return at _11_ is the reserved word.

Code 1

```
1: _1_ include<stdio _2_ _3_ >
2: int _4_ ()
3: {
4:     _5_ number _6_
5:     _7_ ("Enter an integer:") _8_
6:    scanf("%d", _9_ _10_ );
7:    printf("You entered: %d", number);
8:    _11_ 0;
9: }
```

Code 2

```
1: # include<stdio.h>
2: int main()
3: {
4:     int number;
5:     printf("Enter an integer:");
6:    scanf("%d", &number);
7:    printf("You entered: %d", number);
8:    return 0;
9: }
```

3.3 Answer Interface

The answer interface for students has been implemented on a Web browser [5]. The *JavaScript* program verifies the correctness of the answer. Thus, this interface can also be used in offline by storing the necessary files in a client computer. Figure 1 illustrates the interface for this EFP instance. After filling in the answer forms, the student has to click the “Answer” button. If the answer is not correct, the background color of the form will be red. Otherwise, the background color will be white. The student can submit the answers repeatedly until all the answers to be correct.

3.4 Parser

To generate an EFP instance, the source code needs to be separated into a collection of elements with the associated attributes. In this study, the *C parser* is implemented to achieve it.

Figure 2 shows the interaction between the *lexical analyzer* and the *parser*. The parser is also called the *syntax analyzer*. The *lexical analyzer* transforms a given C source code into a sequence of *lexical units* or *tokens* that represent the least elements to compose the code. It can classify each element into either a reserved word, an identifier, a symbol, or an immediate data. The output of the *lexical analyzer* becomes the input to the *parser* for the syntax analysis.

In this study, we originally implement the *lexical analyzer* for a C source code and adopt CUP [17] for the *syntax analyzer*. CUP is an open-source system for generating LALR parsers based on the grammar for which a parser is needed. The output of *CUP* includes *sym.java* and *parser.java* classes. The *sym.java* class contains a series of *constant declarations* for the *symbol table*, and the parser.java class includes a parser itself. Table I shows the associated information contained for each symbol in the *symbol table*. In the implantation of the blank element selection algorithm, the *tokens* from the *lexical analyzer* are used in the vertex generation, and their associated symbol information are used in the edge generation.

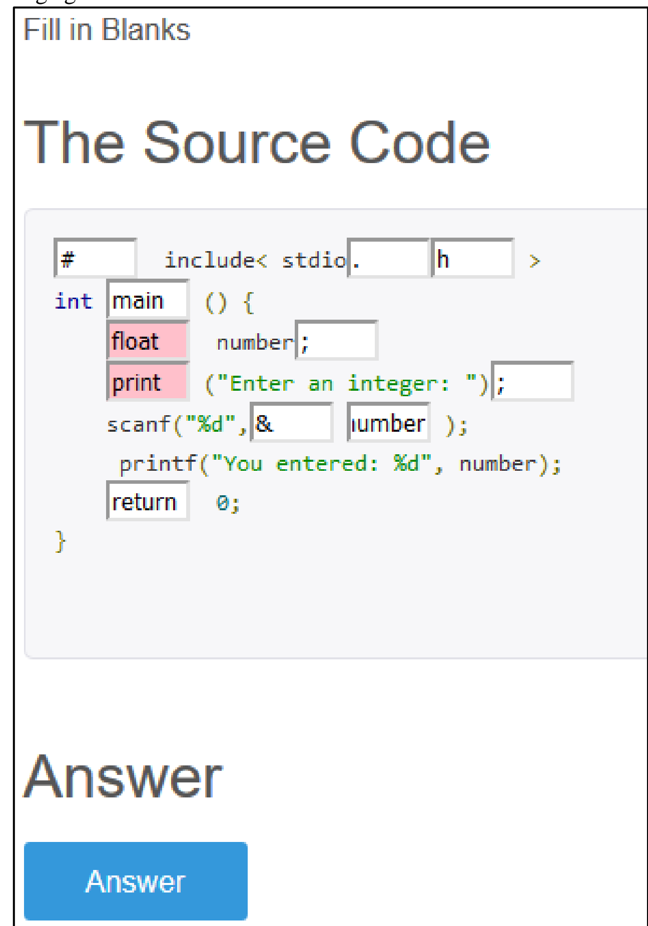

Fig 1　Answer interface.

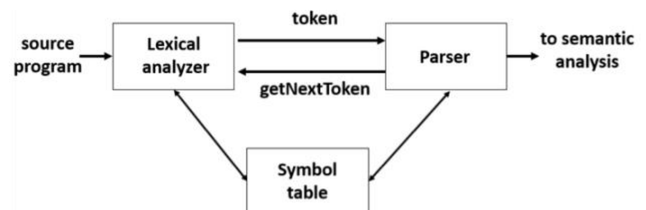

Fig 2　Interaction between lexical analyzer and parser.

4. Applications

In this section, we investigate the effectiveness of the *element fill-in-blank problem (EFP)* in CPLAS at *C programming* educations by analyzing the application results to first year or second year undergraduate students in three universities where one university is in Japan and two universities are in Myanmar.

Copyright © 2021 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

Table 1 Symbol information

item	content
symbol	symbol of element
line	row index of element
column	column index of element
count	number of element appearances
order	appearing order of element in the code
group	statement group index partitioned by { and }
depth	number of { from top

4.1 EFP for Basic Grammar Concepts

Firstly, we generate and evaluate nine EFP instances on basic grammar concepts using source codes in [18].

1) Generated EFP Instances: Table II shows the problem ID (PID), the grammar concept, the number of lines in the source code, and the number of blanks for each EFP instance.

2) Overall Solution Results: First, we analyze and compare the overall solution results to the nine EFP instances on basic grammar concepts by the students in the three universities. Table III shows the number of students, the average and the standard deviation (SD) of the correct answer rates (%), and the average and the standard deviation of the numbers of answer submissions by the students in each university.

Table 2 EFP instances for basic grammar concepts

problem ID (PID)	basic grammar concept	number of lines	number of blanks
1	standard I/O	8	15
2	function, while loop	28	15
3	recursion	17	18
4	file I/O	17	14
5	function	19	16
6	nested loop, pointer	23	20
7	pointer	18	17
8	function	24	20
9	conditional statement	15	14

This table shows that the average correct answer rate increases as the number of answer submissions increases among the three universities at solving the nine EFP instances on basic grammar concepts. It means that students need to continue efforts by revising the answers to the questions until they can correctly solve the instances on basic grammar concepts. If they stop, they cannot achieve high correct answer rates. As a result, the students in *A university* give the lowest average correct answer rate with the least average number of answer submissions. The teacher in *A university* should encourage the students to continue their solving efforts.

Table 3 Overall solution results for basic grammar concepts

Univ.	A	B	C
# of students	25	13	42
ave. of correct rate (%)	87.88	91.42	97.33
SD of correct rate (%)	23.91	26.07	6.03
ave. of submissions	19.68	49.92	64.98
SD of submissions	14.31	38.55	53.75

3) Individual Instance Solution Results: Next, we analyze the solution results of the individual EFP instances by the students. Table IV shows the EFP instance ID, the total number of students in the three universities who did not attempt to solve the corresponding instance, and the average correct answer rate for each university.

This table indicates that the instance with ID = 6 for *nested loop, pointer* topic is most difficult for the students, where six students did not attempt to it. Then, the instance with ID = 8 for function topic follows. In future works, we will improve EFP instances on these topics for better understanding.

The students in A University show the lower average correct rate for any instance, when compared with the students in the other universities. Particularly, they are weak in *pointer* topic at ID=6 and ID=7. The students in C University achieve the high correct rate for any instance. The students in B University perform well except for ID=8, where their correct rates are at the middle between the two universities.

Table 4 Individual instance solution results for basic grammar concepts

instance ID	number of unattempt students	Average correct rate (%)		
		A University	B University	C University
1	0	100%	94%	99%
2	0	94%	94%	98%
3	2	90%	92%	97%
4	2	89%	91%	98%
5	4	84%	92%	98%
6	6	84%	91%	93%
7	4	79%	92%	98%
8	5	86%	84%	98%
9	4	84%	92%	97%
average	3	87.88%	91.42%	97.33%
SD	2.12	6.45%	2.94%	1.73%

4.2 EFP for Data Structures and Algorithms

Secondly, we generate and evaluate 10 EFP instances on data structures and algorithms using source codes in [19].

1) Generated EFP Instances: Table V shows the problem ID, the data structures and algorithms, the number of lines in the source code, and the number of blanks for each EFP instance. When compared with the instances in Table II, they have more numbers of lines and blanks, which suggests they are more difficult.

2) Overall Solution Results: First, we analyze and compare the overall solution results to the 10 EFP instances on data structures and algorithms by the students in the three universities. Table VI shows the number of students, the average and the standard deviation (SD) of the correct answer rates (%) and the average and the standard deviation of the numbers of answer submissions by the students in each university.

This table shows that the average correct answer rate is highest while the number of answer submissions is lowest at *B University.*

Copyright © 2021 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

After the students solved the EFP instances basic grammar concepts, they become familiar to how to solve the EFP instances

Table 5 EFP instances for data structures and algorithms

problem ID (PID)	data structures and algorithms	number of lines	number of blanks
1	stack	54	35
2	queue	73	46
3	hash table	102	45
4	heap data structure	57	32
5	binary tree	47	15
6	breadth first search algorithm	98	46
7	depth first search algorithm	61	26
8	quick sort	38	19
9	longest common sequence	38	23
10	Dijkstra algorithm	48	24

Table 6 Overall solution results for data structures and algorithms

Univ.	A	B	C
# of students	17	19	21
ave. of correct rate (%)	93.14	99.72	95.60
SD of correct rate (%)	11.50	0.55	6.36
ave. of submissions	26.35	21.95	86.71
SD of submissions	19.86	10.20	90.93

for *C programming*, and careful in solving the EFP instances on data structures and algorithms. Again, the students in *A university* give the lowest average correct answer rate with the lower average number of answer submissions than the students in C university. The teacher in *A university* should encourage the students to continue their solving efforts, although they are slightly improved. Besides, the students in *C university* may not have sufficient knowledge and skills in *C programming* including data structures and algorithms, because they submitted answers many times. The teacher in *C university* should carefully observe understanding of the students for C programming.

3) Individual Instance Solution Results: Next, we analyze the solving results of the students in the individual EFP instances for data structures and algorithms. Table VII shows the instance ID, the total number of students who did not attempt to solve the corresponding instance in three universities, and the average correct answer rate for each university.

This table indicates that after solving the EFP instances on basic grammar concepts, the number of unattempt students is reduced, where only one student did not attempt to solve the instance with ID=5 for *binary tree* topic. After solving the EFP instances on basic grammar concepts, the students improved their solving performances to the EFP instances on data structures and algorithms, although these instances assume to be more difficult.

The students in B University achieved almost 100% correct rate for any instance. They studied data structures and algorithms very well. The students in C University achieved over 93% correct rate for any instance. However, the students in A University did not achieve 90% correct rate in two instances. They need more efforts.

Table 7 Individual instance solution results for data structures and algorithms

instance ID	number of unattempt students	Average correct rate (%)		
		A University	B University	C University
1	0	98%	99%	99%
2	0	93%	99%	96%
3	0	90%	99%	94%
4	0	89%	99%	95%
5	1	98%	100%	93%
6	0	95%	100%	96%
7	0	95%	99%	97%
8	0	92%	99%	96%
9	0	91%	100%	94%
10	0	89%	100%	96%
average	0.1	93.14%	99.72%	95.60%
SD	0.32	3.34%	0.43%	1.71%

4.3 Discussions

From the comparisons of the solution results of the students in the three universities, we observed that: 1) for the EFP instances on basic grammar concepts, the average correct answer rate increases as the number of answer submissions increases, 2) for the EFP instances on fundamental data structures & algorithms, the correct answer rate becomes the highest at *B university* where the submission number is the lowest, and 3) for both instances, the students in *A university* give the lowest average correct answer rate with the relatively low average number of answer submissions if compared with the students in other universities.

The observation 1) suggests that the novice students need to continue efforts of modifying and resubmitting the answers to the questions on basic grammar concepts until correctly solving all of them, since they may forget or misunderstand them. The observation 2) suggests that some students can more smoothly solve EFP instances if they carefully read the source codes after solving several EFP instances on basic grammar concepts. The observation 3) suggests that the students in *A University* may have weaker motivations in solving EFP instances for studying *C programming* than the other universities. The teacher needs to encourage them to continue solving them and studying *C programming*.

From the comparisons of the solution results of the individual EFP instances, we observed that: 1) for the EFP instances on basic grammar concepts, the most difficult one is on *pointer*, and 2) for the EFP instances on data structures and algorithms, the solution results are satisfactory for the students in B University and C University.

The observation 1) indicates that the students should more focus on improving the knowledge on *pointer*, where we will study how to assist it in CPLAS. The observation 2) suggests that the students can improve C programming skills by solving EFP instances in CPLAS, where they improved correct rates for more difficult

Copyright © 2021 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

instances on data structures and algorithms after solving EFP instances on basic grammar concepts.

5. Conclusion

This paper investigated the effectiveness of *element fill-in-blank problem (EFP)* for *C programming learning assistant system (CPLAS)* at *C programming educations* by analyzing the application results. 19 EFP instances were generated using C source codes on 1) basic grammar concepts and 2) fundamental data structures & algorithms and were assigned to undergraduate students in one university in Japan and two universities in Myanmar. The results observed that the average correct answer rate increases as the number of answer submissions increases for 1), the rate is highest at one university where the submission number is lowest for 2), and one university gives the lowest rate for both. In future works, we will improve the generation of EFP instances for better understanding and higher motivations of students, generate various EFP instances to cover broad topics, and evaluate their effectiveness in studying C programming through applications to students.

References

[1] https://www.spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020
[2] S. I. Ao et al. ed., IAENG Transactions on Engineering Sciences-Special Issue for the International Association of Engineers Conferences 2016 (Volume II), World Sci. Pub., pp. 517-530, 2018.
[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, May 2017.
[4] H. H. S. Kyaw, N. Funabiki, S. L. Aung, N. K. Dim, and W.-C. Kao, “A study of element fill-in-blank problems for C programming learning assistant system,” Int. J. Edu. and Inf. Tech. (IJIET), vol. 11, no. 6, pp. 255-261, 2021.
[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] A. Kashihara, A. Terai, and J. Toyota, “Making fill-in-blank program problems for learning algorithms,” in Porc. Int. Conf. Comput. Edu., pp. 776-783, 1999.
[7] J. Shinkai, Y. Hayase, and I. Miyaji, “A study of generation and utilization of fill-in-the-blank questions for programming education on Moodle,” IEICE Technical Report, vol. 110, no. 263, pp. 7-10, 2010.
[8] K. Terada and Y. Watanabe, “Automatic generation of fill-in-the-blank programming problems,” in Proc. IEEE MCSoC, pp. 187-193, 2019.
[9] D. Parsons and P. Haden, “Parson’s programming puzzles: a fun and effective learning tool for first programming courses,” in Proc. CRPIT, vol. 52, pp. 157-163, 2006.
[10] P. Brusilovsky and S. Sosnovsky, “Individualized exercises for self-assessment of programming knowledge: an evaluation of QuizPACK,” J. Edu. Res. Comput., vol. 5, no. 6, Sept. 2005.
[11] K. K. Zaw, N. Funabiki, and W.-C. Kao, “A proposal of value trace problem for algorithm code reading in Java programming learning assistant system,” Inform. Eng. Exp., vol. 1, no. 3, pp. 9-18, Sep. 2015.
[12] X. Lu, N. Funabiki, H. H. S. Kyaw, S. L. Aung, and N. K. Dim, “A study of value trace problems for code reading study of C programming,” in Proc. WANC, pp. 445-459, Nov. 2020.
[13] T. Kakeshita, and M. Murata, “Application of programming education support tool pgtracer for homework assignment,” Int. J. Learn. Tech. Learn. Enviro., vol. 1, no. 1, pp. 41-60, 2018.
[14] A. K. Jain, M. Singhal, M. S. Gupta, “Educational tool for understanding algorithm building and learning programming languages,” in Proc. World Cong. Eng. Comput. Sci. (WCECS), pp. 292-295, Oct. 2010.
[15] N. Cavus, H. Uzunbovlu, and D. Ibrahim, “Assessing the success rate of students using a learning management system together with a collaborative tool in web-based teaching of programming languages,” J. Edu. Comput. Res., vol. 36, no. 3, pp. 301-321, 2017.
[16] L. N. de Barros, A. P. dos S. Mota, K. V. Delgado, and P. M. Matsumoto, “A tool for programming learning with pedagogical patterns,” in Proc. OOPSLA Work. Eclipse Tech. Ex., pp. 125-129, Oct. 2005.
[17] CUP, https://czt.sourceforge.net/dev/java-cup/manual.html
[18] http://www.codebind.com/c-examples
[19] https://www.programiz.com/dsa
[20] https://briolliant.org/wiki/pascals-triangle

Copyright © 2021 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.