Report on Practice of a Learning Support System for Reading Program Code Exercise

Takahito Tomoto[1(✉)] and Takako Akakura[2]

[1] Faculty of Engineering, Tokyo Polytechnic University, Tokyo, Japan
t.tomoto@cs.t-kougei.ac.jp
[2] Faculty of Engineering, Tokyo University of Science, Tokyo, Japan

Abstract. Reading the source code of software programs is an effective way of learning, but novice programmers need (1) exercises that involve reading programs by tracing execution manually, and (2) feedback when they interpret the program incorrectly. In this paper, we propose exercises in which students read programs, and we report on the development of a system that provides feedback on mistakes. Furthermore, we also report the results of a comparison, conducted in a laboratory environment, between the approach proposed here and the conventional approach of learning via creating programs, as well as the results of two teaching trials.

Keywords: Learning programming · Learning by reading · Code reading · Tracing

1 Introduction

In courses on computer programming, a common pattern is for the lecturer to first explain syntax or the approach to programming and then present example programs before asking students to create programs in a series of exercises. However, in our experience, reading and understanding other people's programs is just as important as creating programs. The process of reading the source code of a program in this way is known as "code reading", and is regarded as very important.

However, books on the subject typically do little more than describe tips for deciphering programming code. Likewise, most explanations of example code in university courses are conveyed by a one-way flow of information, from teacher to students. This means that students are not explicitly given tasks that involve reading code. In this study, we propose program trace exercises as one type of exercise for reading programs. We also develop and evaluate a learning support system to support these exercises.

2 Program Trace Exercises

2.1 Exercise for Reading Programs

The importance of reading programs is widely recognized, but there are few systems that explicitly require students to read a program and provide feedback on their activities.

S. Yamamoto (Ed.): HIMI 2017, Part II, LNCS 10274, pp. 85–98, 2017.
DOI: 10.1007/978-3-319-58524-6_8

Kanamori et al. [1] and Arai et al. [2] advocate exercises involving reading programs with the goal of understanding the content of the program. Some earlier studies [3] divide the process of creating a program into "algorithm design" (thinking about the flow of processing) and "coding" (converting the flow of processing into the specific expressions of the programming language), but Kanamori et al. propose a reading process that reverses this order. That is, reading can be divided into the process of "decoding" (deciphering the processing flow based on the language-specific expressions) and "meaning deduction" (understanding the goal, or meaning, of the program based on the processing flow). Deconstruction is the inverse of construction above, while comprehension is the inverse of algorithm design (Fig. 1). However, these studies do not present exercises for actually tracing the flow of processing in a program in detail.

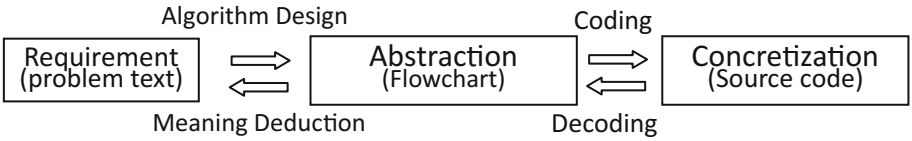

Fig. 1. Program reading process (Kanamori et al.)

To read a program, it is important to learn how to trace "program behavior". This entails working out which line (location) of the program is currently executing, which line will be executed next, and what will happen as a result. Students who cannot accurately trace program behavior in this way will not be able to debug their own programs properly and will end up creating programs with a half-formed understanding. Accordingly, in this study we propose exercises that require students to trace program behavior line-by-line and to think about what kind of instructions are being performed at each line.

2.2 Proposed Program Tracing Exercises

In this section, we explain the program trace exercises proposed by this study, using Fig. 2 for illustration.

To make students think about program behavior, program trace exercises are of two types: (A) exercises asking which line will be executed, and (B) exercises asking about the details of the instructions executed by each line. In these exercises, students are first given some source code. Students are then asked to (A) select which line in the source code will be executed first. Next, students are asked to (B) describe changes in the values of variables or the details of the output. The intent is to have them think about the instructions executed by that line. For loop processing and conditional branching, the flow of program processing changes depending on the outcome of the conditions, and so students are asked to describe the outcome of conditions as well. Thereafter, students are asked to describe the behavior of the program until the program terminates, by selecting which line will be executed next, and so on in the same manner. For programs with a sequential structure, the execution order will be almost the same as the

	(A)	(B)				
ソースコード	実行する行	変数の値			出力	判定
		i	j	a		
void main(){	void main(){					
int i , j , a = 0 ;	int i , j , a = 0 ;			0		
for(i = 1 ; i < 3 ; i++){	for(i = 1 ; i < 3 ; i++){	1				満たす
for(j = 1 ; i < 3 ; j++)	for(j = 1 ; i < 3 ; j++)		1			満たす
a = a + j ;	a = a + j ;			1		
printf("%d¥n", a);	for(j = 1 ; i < 3 ; j++)		2			満たす
}	a = a + j ;			3		
}	for(j = 1 ; i < 3 ; j++)		3			満たさない
	printf("%d¥n", a);				3	
	⋮					

Fig. 2. An example program trace exercise

order in which the lines appear in the source code, but for loops, conditional branching and functions, the processing flow jumps from one line to another, and so these exercises test whether students can trace this processing flow properly. This kind of task is a daily activity for an experienced programmer, but there are beginners who either do not know about this kind of task or who know about it but cannot do it properly and require guidance. For this reason, we believe that it is important to explicitly provide learners with such tasks, and to also provide them with feed-back.

2.3 Previous Research

Yamaguchi et al. [4] developed a system that visualizes which line in the source code is currently executing and changes in the execution environment for that line. Similarly, Yamaguchi et al. [5] visualize differences between the correct program behavior and the program entered by the user as changes in the execution environment. However, these studies go no further than observing behavior, and do not require the learner to generate the behavior themselves. Sugiura et al. [6] and Noguchi et al. [7] require learners to generate behavior consistent with an algorithm in order to develop their ability to create algorithms. However, the goal of these studies is to understand algorithms, and, unlike in this study, they do not ask users to trace behavior based on the source code. Egi et al. [8] have developed a support system that instructs novices in the process of tracing. This system provides support by prompting novices to trace their program when they get stuck creating a program, as well as guidance using trace instructions for diagnosing the state of the system or identifying the location for the solution. However, this system does not conduct traces in the sense of understanding program behavior itself.

3 Learning Support System

3.1 System Overview

As part of this study, we have developed a learning support system with the goal of helping students understand program behavior [9]. The goal of this system is to present learners with program trace exercises, ask them to enter information about program behavior, and then provide them with feed-back. Learners are given source code and asked to (A) select which line will be executed (the execution point) and (B) enter information about the details of the processing executed by that line (such as the values of variables, input and output, and the details of condition evaluations). If the learner's input is incorrect, the system provides feedback designed to prompt them to find the error themselves. Within the range of constraints (A) and (B), this system is not affected by language specifications, but language features outside of this range (such as event-driven programming, pointers and reference passing, objected-oriented programming) are out of scope for this system. We have prepared eight exercises in the C programming language for this system. When users select a problem number, the selected problem is displayed.

3.2 Interface

Figure 3 shows the interface for the system. The system first provides learners with source code (the left-hand side of the screen). Next, learners are asked to click lines in

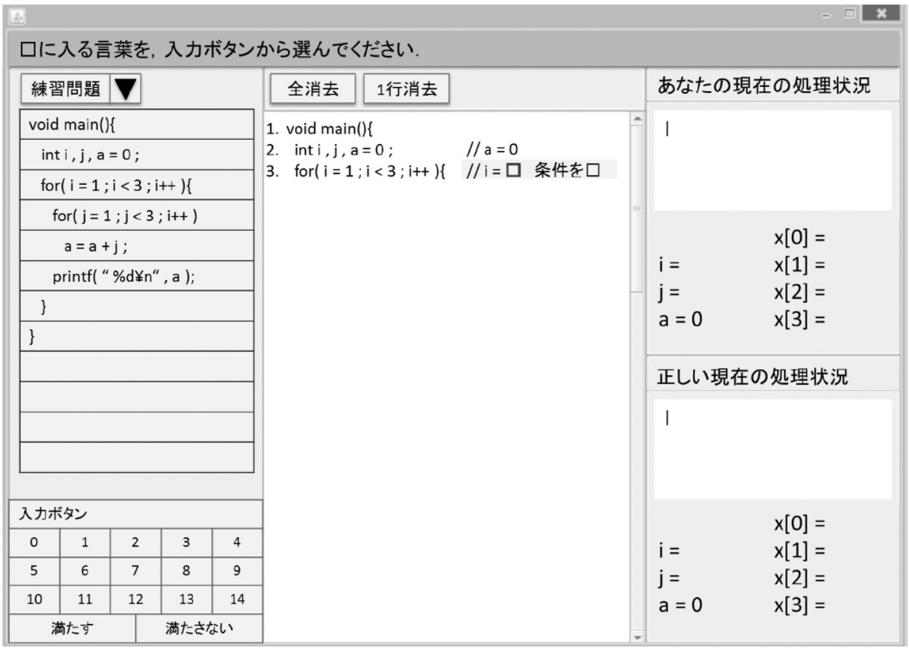

Fig. 3. System interface

the source code in the order in which they are executed (center of the screen). In addition, when processing of constructs such as variable assignments or conditional branching is performed, learners use the input buttons at the bottom left of the screen to enter the details of the corresponding processing.

3.3 Feedback Function

We anticipate that beginners will have errors in understanding of the order of execution and with entering processing details. The program trace exercises themselves can be completed using pen and paper, but when learning independently with pen and paper, or by methods such as looking at reference books, learners may proceed to the next step without realizing that they have made a mistake. When this happens, they will not notice their mistake until they look at the answer at the end of the exercise, if they notice at all. However, they then realize their mistake because they have seen the answer, and so they lose the opportunity to fix the mistake by themselves. Even in cases where exercises are conducted by a teacher in an educational setting, the one-to-many nature of the classroom makes it difficult to expect adequate support. For normal debugging, there is often a trace function, and this can also be used for learning. Many of these debugging functions allow the user to trace program execution by stepping through the program one line at a time, and many also allow users to check the values of variables or the screen output at each line. However, when learning using a debugger, users still find out the correct answer straightaway, and so this approach is still inadequate in terms of learners realizing their mistakes.

In this study, we visualize information input by the user (such as screen output or changes to internal variables that occur in accordance with the program behavior) as "Your current processing state". At the same time, we also visualize the correct screen output and variable values that match the program behavior as "Correct current processing state" (right-hand side of the screen in Fig. 2). If a learner has made a mistake regarding the execution order or the processing for a particular line, then the visualization will show a discrepancy between these two parts of the screen. The learner is asked to check the discrepancy and then identify and fix the error own their own.

Figure 4 shows an example of the visualization content for a learner error. This error is a common mistake that learners make regarding "for" statements. In "for" statements, the processing for the third clause is executed every time the loop repeats but before the continuation condition in the second clause is evaluated as true or false. If the condition is not met, then the loop ends. This means that in the example where the continuation condition is "j < 3", then the loop will end when the value of j becomes 3. However, some learners will see this "for" statement and interpret it vaguely as "repeat with j ranging from 1 to 2". In this case, they will not understand that j is equal to 3 when the loop ends. In this kind of example, a discrepancy will occur in terms of whether the processing for the third clause is performed at the end of the loop or in the part where the condition in the second clause is evaluated. This discrepancy is then visualized as a gap in the timing of the screen output, or a difference in the value of j. Because learners are asked to constantly check these discrepancies in output results or internal variables, they are able to review their answer at the point when the discrepancy occurs.

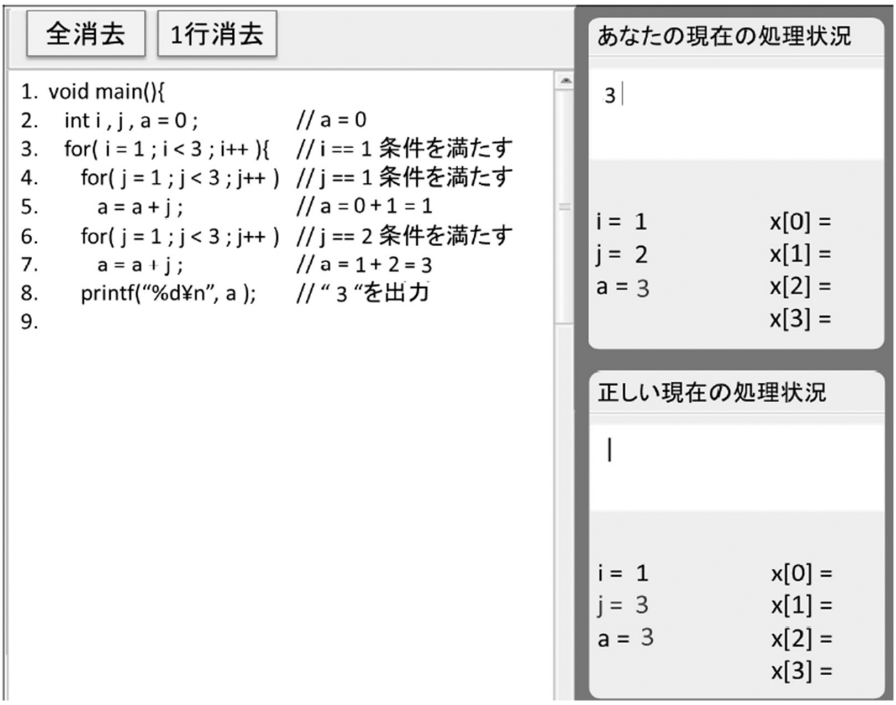

Fig. 4. Feedback from the system

4 Comparison Test

4.1 Test Overview

Objective

We conducted an experiment to assess whether learning with this system promoted better understanding of program behavior than the normal approach of learning by creating programs. In this experiment, we conduct three types of test before and after students used the support system in order to assess whether the system is able to promote more effective program comprehension than normal study. We also assessed whether the proposed program trace exercises and support system were in alignment with the goals of this study, based on subjective assessments by the students (questionnaires).

Test procedure

The test participants were 18 engineering students who had learned the C programing language at university, divided randomly into a test group and a control group with 9 subjects in each group. Both groups took a 30-minute pre-test, followed by 60 min of learning and then a 30-minute post-test and questionnaire. The test group learned via the support system, whereas the control group learned in the conventional manner by creating programs.

Test types

We conducted three types of test for the pre- and post-tests: (1) an output result test, where subjects are asked to write down the output results for given source code, (2) a trace test, where subjects are asked to describe the execution order and variable changes for given source code, and (3) a coding test, where subjects are given output results and source code with blank spaces and asked to write a program by filling in the blanks.

Each test consisted of five problems, with each problem worth one point, for a total of five points. The problems were as follows: (1) source code including nested structures of "for" and "if" statements, (2) source code including double "for" statements, (3) source code including a "for" statement where the control variable is involved in the condition expression, (4) source code including substitutions to the values of arrays and functions, and (5) source including recursive functions.

4.2 Experiment Results

Test results

Table 1 shows the test results, and Table 2 shows the ANOVA results with one between-subjects factor (A: test group versus control group) and two within-subject factors (B: pre-test versus post-test; C: test type). Interactions between A and B were apparent in the ANOVA results, and so we assessed the simple main effects. The results are listed in Table 3, which shows a significant difference between groups for the post-test results ($p < 0.05$). From this, we see that by using the system the test group achieved better results in all three types of test. In particular, there is a clear difference in the post-test results for the trace test and the output result test, implying that learning via the system fosters the ability to trace program behavior. Furthermore, although the control group learned via creating programs, the test group was still able to achieve

Table 1. Results of tests

	Output test			Trace test			Coding test		
	Pre	Post	Difference	Pre	Post	Difference	Pre	Post	Difference
Test group	0.67	2.44	1.77	0.44	2.11	1.67	0.78	2.33	1.56
Control group	0.67	1.22	0.55	0.44	0.78	0.34	0.78	1.44	0.66

Table 2. Results of ANOVA

	Sum of squares	Degrees of freedom	F value	Significance
A: Group	8.90	1	1.76	n.s.
B: Test timing	32.2	1	90.4	<0.01
AB	8.90	1	25.0	<0.01
C: Test type	3.02	2	2.45	n.s.
AC	0.24	2	0.20	n.s.
BC	0.13	2	0.22	n.s.
ABC	0.24	2	0.41	n.s.

Table 3. Results of means for AB interaction

	Sum of squares	Degrees of freedom	F value	Significance
Group (pre)	0.00	1	0.00	n.s.
Group (post)	17.8	1	6.55	<0.05
Test timing (test group)	37.5	1	105.2	<0.01
Test timing (control group)	3.63	1	10.2	<0.01

better results than the control group in the coding test. These results indicate the possibility that students can improve their ability to create proper programs via learning that focuses on tracing program behavior.

Questionnaire result

Table 4 shows the results of the questionnaire given to the test group after they had used the system (four-level multiple choice: 1 Not at all; 2 Not really; 3 A bit; 4 Quite a lot). From these results, we found that participants felt that the visualization of the execution process improved their understanding of the program, and that they reviewed their answers by comparing the visualizations of their own processing state with the correct state. We also found that students felt that their activities using the system led to an understanding of the program. From these results, we found that subjects felt the system and feedback were valuable.

Table 4. Mean answers on questions ($N = 9$)

Do you think that visualization of the execution process and output state makes the program easier to understand?	3.2
If there was a difference between your processing state and the correct processing state, did you think about where you went wrong?	3.3
Do you think your understanding of the program was improved by the system?	3.1

5 Teaching Trial

5.1 Experiment Overview

Objective

In Sect. 4, we found that learning via using the system promoted understanding of program behavior more than normal learning via creating programs. However, only nine participants used the system, the limited assessment took place in a test lab environment, and the entire process from pre-test through to post-test took a bit more than two hours without interruptions. University courses consist of 90-minute time slots, so that conducting an assessment over a similar amount of time would require two time slots, and students may lose concentration. Accordingly, we conducted further experiments to assess whether the system can be used in the context of two normal 90-minute university time slots in an environment where teaching staff cannot thoroughly monitor student activities. We also assessed whether we can expect learning benefits from using the system.

Experimental procedure
Two courses conducted by the Department of Industrial Management and Engineering of the Faculty of Engineering at the Tokyo University of Science were selected as the targets for this experiment: Information Technology Lab 1 (IT1; first semester of second year, 80 students), which is the course where students first study programming and master syntax and elementary coding technology; and Information Technology Lab 3 (IT3; second semester of third year, 66 students), where students with a certain amount of experience studying programming learn about algorithms in greater detail. The timing of the tests was the latter half of each course, namely, July for IT1 and December for IT3. Both of these courses are compulsory for students in the department and are taken by all students, not just those who are particularly good at programming. 73 of the participants in IT1 and all of the participants in IT3 are focusing on the C programming language. In normal classes for these courses, the procedure is for students to learn by creating programs through a series of exercises after first hearing an explanation from the teacher.

For the test procedure, we used a shortened form of the same procedure as was used in the comparison test. However, for the comparison test, we conducted assessments using a 30-minute pre-test, 60 min of learning, and a 30-minute post-test. However, the time slots for the course are 90 min long, meaning that two time slots are required. Accordingly, we decided to conduct the trial by spending the first 30 min of the first time slot on the pre-test, followed by 20 min explaining the system and the lecture content, with the remaining 40 min spent learning via the system. The first 20 min of the second time slot was spent on learning via the system (for a total of 60 min of system usage over the two time slots), followed by a 30-minute post-test and questionnaire. Note that the four teaching assistants in each course were instructed to allow students to work out how to solve problems on their own, only answering questions on how to use the system and responding to technical issues.

Test types
Three types of test were conducted for the pre- and post-tests, in the same manner as for the comparison test: (1) an output result test, where participants are asked to write down the output results for given source code; (2) a trace test, where participants are asked to describe the execution order and variable changes for given source code; and (3) a coding test, where participants are given output results and source code with blank spaces and asked to write a program by filling in the blanks. The problems and marking methods were also the same as for the comparison test.

5.2 Experimental Results for Information Technology Lab 1

Test results for Information Technology Lab 1
Table 5 shows the test results for when the system was used in IT1 (first semester of second year), consisting mainly of students with no programming experience. Table 6 shows the ANOVA results with two within-subject factors (A: pre-test versus post-test; B: test type [output result, tracing or coding]). As seen in Tables 5 and 6, we found a significant difference in test timing (pre-test vs. post-test), showing that using the system promotes students' understanding. Moreover, Table 7 shows the results for the

simple main effects in the interaction between factors A and B. In Tables 5 and 7, significant differences are apparent between test types (output results, tracing and coding) in the pre-tests, and so we conducted multiple comparisons for test type in the pre-test results. The results, summarized in Table 8, show significant differences between the tracing test and the other tests. From this, we found that the novice programmers taking IT1 could, as a result of learning via the system, improve their ability to predict output based on given source code, as well as their ability to create appropriate source code based on required output and their ability to understand program behavior based on source code. In particular, we found that the ability to properly understand program flow improved as a result of the system, despite being weaker than the other skills at the time of the pre-test.

Table 5. Results of tests for IT1

	Output test			Trace test			Coding test		
N = 80	Pre	Post	Difference	Pre	Post	Difference	Pre	Post	Difference
IT1	0.95	1.35	0.40	0.33	1.23	0.90	0.76	1.34	0.58

Table 6. Results of ANOVA for IT1

	Sum of squares	Degrees of freedom	F value	Significance
A: Test timing	46.9	1	1.76	<0.01
B: Test type	12.1	2	90.4	<0.01
AB	5.15	2	25.0	<0.01

Table 7. Results of means for A*B interaction for IT1

	Sum of squares	Degrees of freedom	F value	Significance
Test timing (output)	6.40	1	15.8	<0.01
Test timing (trace)	32.4	1	80.0	<0.01
Test timing (coding)	12.2	1	32.7	<0.01
Test type (pre)	16.5	2	14.9	<0.01
Test type (post)	0.76	2	0.69	n.s

Table 8. Means on factor B (a1) for IT1

	Nominal significance level	*t* value	Significance
Output-trace	0.017	5.32	<0.01
Output-coding	0.033	1.60	n.s.
Coding-trace	0.033	3.72	<0.01

Questionnaire results for Information Technology Lab 1

Table 9 shows the results of the questionnaire given to the students taking IT1 (four-level multiple choice, as before). The results obtained are positive, as was the case for the comparison test. The students taking IT1 are programming novices, but we

Table 9. Mean answers on questions for IT1 (N = 80)

Do you think that visualization of the execution process and output state makes the program easier to understand?	3.3
If there was a difference between your processing state and the correct processing state, did you think about where you went wrong?	3.3
Do you think your understanding of the program was improved by the system?	3.0

found that they believe that their understanding of programming has improved as a result of their learning activities using the system, and that they feel that their understanding of the programs has improved as a result of the visualization of the execution process. We also found that they reviewed their answers by comparing their own processing state with the correct state. From these results, we found that learning via system and the feedback provided by the system were received as valuable, even by novices with no programming experience.

5.3 Experimental Results for Information Technology Lab 3

Test results for Information Technology Lab 3

Table 10 shows the test results for when the system was used in IT3 (second semester of third year), which is a course for students who have already studied a certain amount of programming to develop a deeper understanding of more advanced algorithms. Table 10 shows the ANOVA results with two within-subject factors (A: pre-test versus post-test; B: test type [output result, tracing or coding]). As seen in Tables 10 and 11, we found a significant difference in test timing (pre-test versus post-test), just as for IT1. From these results, we infer that the system has learning benefits for students who have already studied some programming and algorithms. Moreover, Table 12 shows the results for the simple main effects in the interaction between factors A and B. In Tables 10 and 12, significant differences are apparent in the test timing (pre-test versus post-test) for all test types. In addition, significant differences are also apparent between test types (output results, tracing and coding) in the pre-tests, and so we conducted multiple comparisons for test type in the pre-test results. The results, summarized in Table 13, show significant differences between the coding test and the other tests. One possible reason that the scores for the coding test were lower than for the other tests is that the goal of IT3 is for students to understand algorithms, rather than to learn coding as such. A certain amount of study on tracing programs and predicting output results may have already been covered in the course.

Table 10. Results for IT3

	Output test			Trace test			Coding test		
N = 66	Pre	Post	Difference	Pre	Post	Difference	Pre	Post	Difference
IT3	1.02	1.85	0.83	1.05	1.77	0.73	0.65	1.95	1.30

Table 11. Results of ANOVA for IT3

	Sum of squares	Degrees of freedom	F value	Significance
A: Test timing	90.2	1	104.2	<0.01
B: Test type	1.25	2	0.58	n.s.
AB	6.20	2	5.94	<0.01

Table 12. Results of means test for A*B interaction

	Sum of squares	Degrees of freedom	F value	Significance
Test timing (output)	22.9	1	36.0	<0.01
Test timing (trace)	17.5	1	27.4	<0.01
Test timing (coding)	56.0	1	88.0	<0.01
Test type (Pre)	6.34	2	3.98	<0.05
Test type (Post)	1.10	2	0.69	n.s

Table 13. Means on factor B (a1)

	Nominal significance level	*t* value	Significance
Output-trace	0.033	0.195	n.s.
Output-coding	0.033	2.34	<0.05
Coding-trace	0.017	2.54	<0.05

Table 14 summarizes the test results for IT1 and IT3. In this study, we used the same tests for both courses, and so the pre-test results for the more advanced IT3 students tend to be better. Moreover, we also found that the improvement in scores was more apparent for IT3. This result seems to indicate that the system may be more effective for students who are more advanced and who have a better understanding of algorithms.

Table 14. Comparison of results of tests for IT1 and IT3

	Output test			Trace test			Coding test		
	Pre	Post	Difference	Pre	Post	Difference	Pre	Post	Difference
IT1	0.95	1.35	0.40	0.33	1.23	0.90	0.76	1.34	0.58
IT3	1.02	1.85	0.83	1.05	1.77	0.73	0.65	1.95	1.30

Questionnaire results for Information Technology Lab 1

Table 15 shows the results of the questionnaire given to the students taking IT3 (four-level multiple choice, as before). The results obtained are positive, as high or higher than the questionnaire results for IT1. From these results, we see that the participants in IT3, who are somewhat more advanced in their study of programming and algorithms, have a greater recognition of the value of the system, feedback and learning method than the novice programmers in IT1.

Table 15. Mean answers on questions for IT3 (N = 66)

Do you think that visualization of the execution process and output state makes the program easier to understand?	3.6
If there was a difference between your processing state and the correct processing state, did you think about where you went wrong?	3.6
Do you think your understanding of the program was improved by the system?	3.5

6 Conclusion

In this study, we proposed exercises for learning programming that require students to actually trace the behavior of program processing. Using this system appears to be beneficial for learning programming, and the questionnaire results show positive responses, indicating that students accept the system.

Here, for the sake of input simplicity, we have asked students to trace behavior one line at a time. However, a "for" statement, for instance, includes multiple expressions in a single line, and it is important that students properly understand the order of execution of these expressions. In future, we aim to support program tracing at the level of individual expressions.

Acknowledgment. Part of this study was funded by a Grant-in-Aid for Scientific Research, Basic Research (C) (10508435).

References

1. Kanamori, H., Tomoto, T., Kometani, Y., Takako, A.: Proposal for 'Learning via Reading Programs' in the programming process and development of a learning support system for the 'Comprehension' process. Trans. Inst. Electron. Inf. Commun. Eng. Jpn. **J97-D**(12), 1843–1846 (2014)
2. Arai, T., Kanamori, H., Tomoto, T., Kometani, Y., Akakura, T.: Development of a learning support system for source code reading comprehension. In: Yamamoto, S. (ed.) HIMI 2014. LNCS, vol. 8522, pp. 12–19. Springer, Cham (2014). doi:10.1007/978-3-319-07863-2_2
3. Shinkai, J., Sumitani, S.: Development of programming learning support system emphasizing process. Jpn. Soc. Educ. Technol. **31**(Suppl.), 45–48 (2007). (in Japanese)
4. Yamashita, K., Nagao, T., Kogure, S., Noguchi, Y., Konishi, T., Ito, Y.: An educational practice using a code reading support environment for understanding nested loop. IEICE Tech. Rep. **114**(82), 7–12 (2014). (in Japanese)
5. Yamoto, R., Noguchi, Y., Kogure, S., Yamashia, K., Konishi, T., Ito, Y.: A learning environment for teaching students how to debug systematically. In: Proceedings of the 39th National Convention, Japanese Society for Information and Systems in Education, pp. 453–454 (2014). (in Japanese)
6. Sugiura, M., Matsuzawa, Y., Okuda, K., Ohiwa, H.: Introductory education for algorithm construction: understanding concepts of algorithm through unplugged work and its effects. J. Inf. Process. **49**(10), 3409–3427 (2008). (in Japanese)

7. Noguchi, Y., Nakahara, T., Konishi, T., Kogure, S., Itoh, Y.: Construction of a learning environment for algorithm and programming where learners operate objects in a domain world. Int. J. Knowl. Web Intell. **1**(3–4), 273–288 (2010)
8. Egi, T., Takeuchi, A.: Development and evaluation of debugging support system of guide tracing for beginners. Jpn. J. Educ. Technol. **32**(4), 369–381 (2009). (in Japanese)
9. Tomoto, T., Asai, K., Tamura, Y., Akakura, T.: Development and evaluation of learning support system for programming reading exercise. IEICE Tech. Rep. **115**(50), 7–10 (2015). (in Japanese)