# Proposal of a Method for Generating C Program Tracing Tasks


1st Yuichiro TATEIWA
*Nagoya Institute of Technology*
Aichi-ken, Japan
tateiwa@nitech.ac.jp

2nd Tomohiro MOGI
*Chiba Institute of Technology*
Chiba-ken, Japan
tomo1238hi@gmail.com

3rd Takahito TOMOTO
*Chiba Institute of Technology*
Chiba-ken, Japan
tomoto@net.it-chiba.ac.jp

4th Takako AKAKURA
*Tokyo University of Science*
Tokyo-to, Japan
akakura@rs.tus.ac.jp



***Abstract*—This abstract represents our full paper categorized under the 'Research to Practice' category.**

**Program tracing tasks ask about the behavior of a program, such as "the next expression to be executed" and "the value of variables during the execution of that expression". The program tracing exercise system developed by Tomoto et al. provides an environment for efficiently conducting exercises using program tracing tasks. This system poses questions to learners based on program tracing tasks, adapting to their answers.**

**However, implementing questions for new programs in this system requires considerable effort and time. Therefore, we propose a method to automatically generate tracing tasks by identifying the targets of questions from the program and following predefined question guidelines. Specifically, we first create a model that formalizes the tracing tasks. This model consists of the targets of questions, their execution order, and execution conditions. Then, we develop a method to generate data from the program to implement this model. This method derives an abstract syntax tree from the program, identifies question targets based on the attribute values of nodes, and specifies the order and conditions for executing questions based on the relationships between nodes. Consequently, tracing tasks can be generated by implementing the model with the generated data. We found that tracing tasks could be generated using the proposed model and the data extracted by the proposed method from programs extracted from a programming exercise textbook.**



***Index Terms*—program tracing task, e-learning, programming**


## I. Introduction

In recent years, there has been a growing demand for programming education due to the introduction of mandatory programming in elementary school education and an increased awareness of the importance of computational thinking. One of the crucial aspects of this education and computational thinking is not merely understanding the language of programs but also comprehending algorithms and program behavior (how they work). On the other hand, many programming beginners have a vague understanding, such as "for loops are used for multiple iterations." As a result, they may have an ambiguous understanding of whether the condition in a for loop is a continuation condition or a termination condition, or they may not fully grasp the state of the iterator when exiting a loop.

To ensure a thorough understanding of program behavior, rather than having a vague idea, program tracing tasks (hereafter referred to as tracing tasks) proposed by Tomoto et al. [1] become crucial. Tracing tasks ask about the parts of a program that are executed, their order, and the execution details at each part. Moreover, Tomoto et al. developed an interactive program tracing exercise system called LARC (**L**earning **A**rchitecture for **R**eading **C**omprehension of Programming) [1] that poses questions to learners based on tracing tasks and provides feedback based on their answers. The feedback consists of the values of variables and standard output based on the learner's answer and the correct answer. By reflecting on the differences in variable values and standard output from their own answers, learners become aware of their misunderstandings regarding program behavior.

On the other hand, conducting exercises with new programs using LARC requires the following tasks, which can be a significant burden for teachers:

**Step 1)** The teacher analyzes the program to find the targets of questions, determines the execution order and conditions for questions, and implements them in LARC.

**Step 2)** The teacher traces the program to determine the values of variables and standard output, and implements them in LARC to display them according to the questions.

**Step 3)** To show the execution results based on the answers, the teacher implements the same operations in LARC for those parts.

Therefore, we developed a system that generates tracing tasks from programs and provides exercises similar to LARC using these tasks. We then assessed how well tracing tasks can be generated from programs in textbooks [2]. However, the method for generating tracing tasks in this system included temporary and provisional elements and was not established enough to be made public.

An abstract syntax tree is a tree structure that represents each component of a program (variables, functions, control structures, etc.) as a node. Tracing tasks ask about the expressions executed within user-defined functions or the values of variables that constitute those expressions. It is believed that these questions can be generated based on the data obtained by traversing the abstract syntax tree.

This research aims to streamline Step 1 and begins by proposing a model that formalizes tracing tasks. This model consists of the targets of questions, their execution order, and execution conditions. We then propose a method to generate data from the program to implement this model. This method

derives an abstract syntax tree from the program, identifies question targets based on the attribute values of nodes, and specifies the execution order and conditions for questions based on the relationships between nodes. Consequently, tracing tasks can be generated by implementing the model with the generated data.

## II. Related work

This chapter discusses the extent to which existing tools and systems can be used for program tracing exercises.

The system [3] presents randomly generated C programs under specified conditions and can ask about the post-execution values of variables. This question is a part of tracing tasks, but it does not ask about other aspects (for example, the values of variables during program execution).

LEPA [4] fragments a given C language program and adds code to observe the execution of each fragment. It then executes the program to obtain the execution history. Afterward, it considers some of the execution results of each fragment as operations (concepts that help learners understand algorithms, expressed as compare A with B or swap A for B, for example) and can present the operations, the source code fragments corresponding to the operations, and the values of variables before and after the execution of those fragments. In this way, LEPA is useful for code reading of C language programs but cannot generate tracing tasks.

CIMEL [5] has the function of presenting a Java program and asking learners about the value of a certain variable at a certain execution point. For example, it can conduct exercises that encourage learners to trace how array values change within a loop. This question is a part of tracing tasks, but it does not ask about other aspects (for example, the execution order of program expressions). Moreover, it does not have the function to automatically generate questions.

ChiQat-Tutor [6] has the function of asking about the behavior of recursive programs. Learners answer recursive calls in the behavior of recursive programs and the processing statements before and after the calls. This question can be considered a type of tracing task, but its targets are limited to instructions strongly related to recursive calls. In addition, it does not have the function to automatically generate questions.

The GNU Debugger (GDB) [7] can single-step execute a C language program and display the values of variables and screen output at that point. GDB is useful for obtaining the correct answers to tracing tasks but cannot generate tracing tasks.

## III. Program tracing exercise

This chapter discusses the program tracing tasks, the LARC system that realizes an environment for efficiently practicing these tasks, and the learning effects of exercises using LARC, all of which were proposed by Tomoto et al. in their research [1].

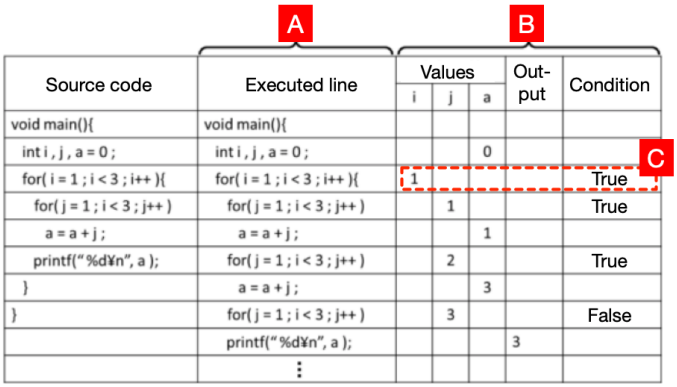


Fig. 1. Example of program tracing task.

### A. Program tracing tasks and feedback

Tracing tasks ask learners about the order of executed lines (A in Figure 1) and the details of instructions executed on each line (B in Figure 1) during program execution to engage them in thinking about the program's behavior. In the case of sequential structures, the order of lines in the source code and the execution order are the same. However, in the case of iteration, conditional branching, and functions, line jumps occur, so A is asked. In the example in the figure, the learner answers '`void main(){`' as the first executed line, '`int i, j, a = 0;`' as the next executed line, and '`printf("%d\n", a);`' as the ninth executed line. B is asked to make learners think about the instructions executed on the executed lines. Especially in iterative processing and conditional branching, the flow of program processing changes depending on the success or failure of conditions, so B also asks about the success or failure of conditions. In C of the figure, the learner answers 1 as the value assigned to variable i, and True as the value of the condition expression '`i < 3`'.

Tomoto et al. believed that for deepening understanding, it was important for learners to arrive at the correct answer themselves, rather than easily presenting the correct answer to tracing tasks. Therefore, they proposed providing displays (feedback) that encourage learners to notice their own mistakes, according to their answers. Feedback appropriately displays the standard output and variable values based on the learner's answer in parallel with the standard output and variable values based on the correct answer. If the learner's answer is incorrect, a difference will occur between the two displays. Learners are expected to detect this difference, identify the error location, and correct it.

### B. LARC

LARC is a system that realizes an environment for efficiently conducting exercises using program tracing tasks. It can present tracing tasks to learners, have them input the program's behavior, and provide feedback based on their input.

Figure 2 shows the execution screen of LARC while answering a tracing task. Learners click and select the lines to be executed from the source code on the left side of the screen.

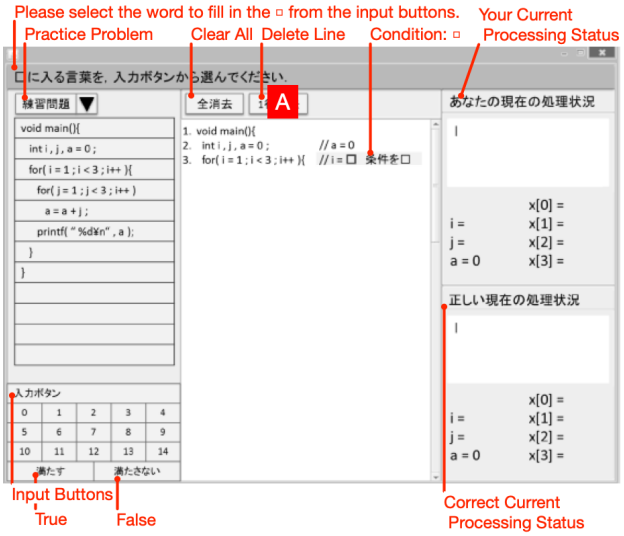


Fig. 2. Example of execution screen in LARC

The selected lines are displayed in the order of selection in the center of the screen. If an assignment, value calculation, standard output, or conditional branch occurs in the selected line, LARC displays input fields for each question in the center of the screen, and learners answer using the input buttons at the bottom left of the screen.

The right side of the screen is an area for feedback, and its display is updated according to the learner's input. The upper half displays the standard output and variable values generated based on the learner's answer, while the lower half displays the standard output and variable values based on the correct answer. If the learner incorrectly answers the lines to be executed or the processing for each line, there may be differences between the two displays above and below. To correct their answer, learners can return to previous questions by pressing the button A.

In the figure, the learner answers the first line of the source code as the first executed line. Then, he/she answers the second line as the second executed line, and 0 as the value of variable a at that time. Consequently, 'a = 0' is displayed in the upper half of the right side of the screen. Also, the value of a based on the correct answer, which is 0, is shown in the lower half of the right side of the screen. Next, the learner answers the third line as the third executed line, and LARC asks about the value of variable i and the value of the condition expression '`i < 3`'.

### C. Learning effects of LARC

The learning effects of LARC were measured through a controlled experiment with 18 engineering university students who had previously learned the C language as participants. The experimental group and the control group engaged in a pre-test for 30 minutes, learning for 60 minutes, a post-test for 30 minutes, and a questionnaire, in that order. During the learning phase, the experimental group worked on tracing tasks with feedback using LARC, while the control group worked on conventional programming tasks.

The tests consisted of three types: an output result test (a test where the source code is given, and participants are asked to fill in the output result), a tracing test (a test where the source code is given, and participants are asked to describe the execution order and variable changes), and a description test (a test where the output result and source code with blanks are given, and participants are asked to write the program that fits in the blanks). The scope of each question included "source code containing nested structures of for and if statements," "source code containing double for loops," "source code containing for loops where the control variable is involved in the condition expression," "source code containing function and array value swapping," and "source code containing recursive functions."

The experimental group performed better in all three types of tests. Notably, although the control group created programs in the programming tasks, the experimental group was able to outperform the control group even in the results of the description test. The experimental results suggested that exercises using LARC may improve not only the ability to trace the behavior of programs but also the ability to create appropriate programs.

## IV. Extension of Program Tracing Exercise

With the aim of further enhancing learning effectiveness, we extend the program tracing exercise described in Section III. To this end, we redefine the learning objectives and redesign the exercise procedure for these learning objectives.

### A. Learning targets

In this research, we define the program behavior that learners should understand through tracing tasks as follows:

- Program start/end: When starting a program, the execution environment executes the main function. When the execution environment executes the last instruction of the main function or a return statement, the program ends.
- Variable declaration and initialization: The execution environment allocates memory based on the declaration and sets values through initialization.
- Variable reference: During the execution of instructions or functions, the execution environment references the values of related variables. Variables have a scope within the program where they can be referenced. Shadowing may occur when a variable declared within a scope hides a variable with the same name already declared in an outer scope. In this case, the outer variable cannot be directly referenced within the inner scope, and the inner variable is referenced instead.
- Operations: The execution environment performs arithmetic operations (e.g., addition, subtraction), logical operations (e.g., AND, OR), relational operations (e.g., equal to, greater than), and assignment operations (setting a specific value to a variable) based on the operators.

- Function call: When calling a function, the execution environment copies the values of the actual data or variables passed to the function (actual arguments) to the arguments specified in the function header of the function definition (formal arguments). It then starts executing from the first instruction of the called function.
- Function return: When the execution environment executes a return statement within a function or reaches the last instruction of the function, it determines the return value from the function. It then resumes execution from the next instruction of the caller, and the return value becomes accessible from the caller.
- Execution of control statements: When executing control statements (e.g., if statements, for statements), the execution environment determines the next instruction to execute based on the condition expression or evaluation expression. Additionally, during the execution of a for statement, the initialization expression and increment expression are executed at predetermined timings.
- Array operations: The execution environment references the value of elements based on the index, assigns values to elements based on the index, or obtains the starting address of the array.

### *B. Exercise procedure in program tracing exercise system*

The exercise in the program tracing exercise system progresses through interaction between the system and the learner, repeating the cycle of "system asks $\rightarrow$ learner answers $\rightarrow$ (if necessary) system provides feedback." Procedures $P$ and $Q$ define this interaction, and the interaction begins with the execution of $P$.

*1) Procedure $P()$:* Procedure $P()$ mainly asks about the next element (function or expression) to be executed and its execution result.

**P1)** The system asks about the user-defined function to be executed by presenting the user-defined functions in the program as options. The learner answers this question by selecting one of the options (let the answer be $ud$).

**P2)** The system asks about the values of the arguments of $ud$ in order from the first argument by presenting the argument names. The learner answers each question by entering the values.

**P3)** The system asks about the expression to be executed by presenting the expressions and function returns in $ud$ as options. The learner answers this question by selecting one of the options (denote the answer as $e$).

**P4)** If $e$ is a function return, proceed to P7; otherwise, the system executes procedure $Q(e)$.

**P5)** If $e$ is a condition expression of an if statement or similar, the system asks about the truth value of $e$ by presenting the return value of $Q(e)$. The learner answers this question by entering the boolean.

**P6)** If $e$ is the argument of a return statement, the system asks about the value of $e$ by presenting the return value of $Q(e)$. The learner answers this question by entering the value (denote the answer as $rv$).

**P7)** The system updates the display of 'standard output' and 'variable values' based on the correct answer (feedback).

**P8)** If $e$ is not a function return, return to P3.

**P9)** The system asks about the return destination from $ud$ by presenting the callers of $ud$ in the program as options. The learner answers this question by selecting one of the options (denote the answer as $cl$).

**P10)** The system sets the return value to $rv$ and $cl$.

*2) Procedure $Q(e)$:* Procedure $Q(e)$ mainly extracts the targets of questions (e.g., variables) by recursively decomposing the expression $e$ received from procedure $P()$ based on the operators and asks questions specific to each element. Procedure $Q(e)$ with expression or expression element $e$ performs the interactions defined in $Q1$ to $Q7$ below according to $e$ and finally returns to its caller via $Q8$. To improve exercise efficiency and avoid learner confusion, some questions are omitted in $Q5$.

**Q1)** If $e$ is an assignment expression, the system executes procedure $Q(r)$ with the right-hand side as $r$, executes procedure $Q(l)$ with the left-hand side as $l$, and asks about the value to be assigned to the left-hand side by presenting the return values of each. The learner answers this question by entering the value. Finally, the system updates the display of 'variable values' based on the answer (feedback).

**Q2)** If $e$ is a binary operation, the system executes procedure $Q(l)$ with the first operand of the operator with the highest precedence as $l$ and executes procedure $Q(r)$ with the second operand as $r$. If the instructor has specified this operator based on the exercise policy, the system calculates the expression based on the return values of each.

**Q3)** If $e$ is an increment or decrement operation, the system executes procedure $Q(o)$ with the operand as $o$. Then, it asks about the value after the operation by concretizing $e$ based on the return value and presenting it (note that the value before the operation is asked in Q5). The learner answers this question by entering the value.

**Q4)** If $e$ is an array reference, the system executes procedure $Q(l)$ with the first operand of the operator with the highest precedence as $l$ and executes procedure $Q(r)$ with the second operand as $r$. If the operator with $e$ as the first operand is not an array operator, the system asks about the value of $e$ by concretizing $e$ based on the return values of $Q(l)$ and $Q(r)$ and presenting it. For example, in $a[b][c]$, if $e$ is $a[b][c]$, it asks about the value of $e$, and if $e$ is $a[b]$, it does not ask about the value of $e$. Finally, the learner answers this question by entering the value.

**Q5)** If $e$ is a variable and does not meet the following conditions, the system asks about the value of $e$ by presenting the variable name. The learner answers this question by entering the value.

- $e$ is the left-hand side of an assignment operator (meaning it does not ask about the variable value on the left-hand side before the assignment execution)

- $e$ is the array name of an array reference (outside the learning scope)

**Q6)** If $e$ is a library function, the system executes procedure $Q(p)$ with each argument as $p$ from the first argument, and the system calculates $e$ based on their return values. Then, the system updates the display of 'standard output' based on the calculation result (feedback).

**Q7)** If $e$ is a user-defined function, the system executes procedure $Q(p)$ with each argument as $p$ from the first argument and executes procedure $P()$. Then, it updates $e$ based on the return value $cl$ of $P()$ (because the caller and return destination may differ due to incorrect answers) and considers the return value $rv$ of $P()$ as the value of $e$.

**Q8)** The system concretizes $e$ based on the answers and calculation results obtained in Q1-Q7 and returns it as the return value.

## V. Realization approach

The questions generated by the system described in Section IV (including their execution order and conditions) serve as the model for tracing tasks. In this research, we generate the data necessary for implementing the model from programs by utilizing the Abstract Syntax Tree (AST) of Clang [**?**].

### A. Clang's abstract syntax tree

An abstract syntax tree (AST) is a tree structure that represents each component of a program (variables, functions, and control structures) as a node. This allows capturing the hierarchical structure of the program. For example, a function definition exists as a single node in an AST, and the variable declarations and instructions within it are associated as its children. The proposed tracing tasks ask about the expressions executed within user-defined functions or the values of variables that constitute those expressions. It is believed that these questions can be generated based on the data obtained by traversing the tree structure of the AST.

Clang is developed as part of the LLVM project, and its quality and performance have attracted the attention of many developers and companies. Clang is an open-source project, and continuous development and testing are carried out by numerous developers worldwide. As a result, bug fixes and performance optimizations are performed regularly, maintaining high reliability. Moreover, Clang is designed to adhere to standard specifications, allowing accurate syntax and semantic analysis.

Clang is widely recognized and used in many development environments and toolchains. It is particularly popular as a compiler for C, C++, and Objective-C, and is adopted as the standard compiler in Apple's Xcode and many Linux distributions. Furthermore, various tools, such as code analysis tools and editor plugins, are developed using the API provided as Clang's library. This allows Clang's AST to be utilized for a wide range of purposes, including static analysis, code generation, and refactoring.

In this research, we use the JSON-formatted AST generated by Clang. Our investigation shows that this JSON object corresponds to the nodes of the AST. The next section describes the implementation using the following attributes:

- 'kind' represents the type of node.
- When 'storageClass' is extern, it means that the section (variable or function) is defined in another file.
- 'inner' is an array that stores the child nodes.
- 'referencedDecl' is the declaration node of the node (variable or function).
- 'id' is the identifier of the node.
- 'opcode' is a string representing the operator of the node.

### B. Data generation based on AST

We generate the data necessary for procedures $P()$ and $Q(e)$ from the program by traversing the AST as follows. In the AST node $node$, the function $dfs(node)$ is a function that evaluates $node$ and then recursively performs depth-first search by calling $dfs(node.inner[i])$ on $node.inner[i]$, where $i$ is the index of the child. Also, when an element $e$ (e.g., a variable) in the program is managed by the node $node$ in the AST, the function $ast(e)$ returns $node$. The realization of feedback (P7, Q1, Q6) is not the purpose of this research, and P4, P8, P10, Q8 are self-evident, so their explanations are omitted.

**For P1)** A user-defined function in the program is a node $node$ in the AST where $node.kind$ is $FunctionDecl$ and $node.storageClass$ is $extern$.

**For P2)** The $i$-th argument of $ud$ is a descendant of $F$ and $F.inner[i-1].kind$ is $ParmVarDecl$ where $F = ast(ud)$.

**For P3)** Each expression in $ud$ is a node $node$ traversed by $dfs(F.inner[i])$ where $F = ast(ud)$ and $F.inner[i].kind$ is $CompoundStmt$, and $node.kind$ is $BinaryOperator$, $UnaryOperator$, $CompoundAssignOperator$, $CallExpr$, $ReturnStmt$, or $VarDecl$ and $node.hasOwnProperty(\text{`}inner\text{'})$ is true, excluding its descendants.

**For P5)** For example, when $e$ is the condition expression of an if statement, in $F = ast(ud)$, $e$ is $node.inner[0]$ of the node $node$ where $node.kind$ is $IfStmt$ traversed by $dfs(F.inner[i])$. Also, when $e$ is the condition expression of a for statement, $e$ is $node.inner[2]$ of the node $node$ where $node.kind$ is $ForStmt$.

**For P6)** When $e$ is the argument of a return statement, in $F = ast(ud)$, $e$ is $node.inner[0]$ of the node $node$ where $node.kind$ is $ReturnStmt$ traversed by $dfs(F.inner[i])$.

**For P9)** A caller of $ud$ in the program is a node $node$ in the AST where $node.kind$ is $CallExpr$ and $node.inner[0].inner[0].referencedDecl.id$ matches $F.id$ where $F = ast(ud)$.

**For Q1)** When $e$ is an assignment expression, in $node = ast(e)$, $node.kind$ is $BinaryOperator$ and $node.opcode$ is $=$, or $node.kind$ is

$CompoundAssignOperator$. $r$ is the subtree rooted at $node.inner[1]$, and $l$ is the subtree rooted at $node.inner[0]$.

**For Q2)** When $e$ is a binary operation, in $node = ast(e)$, $node.kind$ is $BinaryOperator$. $l$ is the subtree rooted at $node.inner[0]$, and $r$ is the subtree rooted at $node.inner[1]$. Whether it is a specific operator is determined by the value of $node.opcode$.

**For Q3)** When $e$ is an increment or decrement operation, in $node = ast(e)$, $node.kind$ is $BinaryOperator$, and $node.opcode$ is $++$ or $--$.

**For Q4)** When $e$ is an array reference, in $node = ast(e)$, $node.kind$ is $ArraySubscriptExpr$. $l$ is the subtree rooted at $node.inner[0]$, and $r$ is the subtree rooted at $node.inner[1]$. When the operator with $e$ as the first operand is an array operator, in the AST node $node2$, $node2.inner[0]$ is $node$, and $node2.kind$ is $ArraySubscriptExpr$.

**For Q5)** When $e$ is a variable, in $node = ast(e)$, $node.kind$ is $DeclRefExpr$. When $e$ is on the left-hand side of an assignment operator, in the AST nodes $node$ and $node2 = ast(e)$, $node2$ is $node.inner[0]$, $node.kind$ is $BinaryOperator$, and $node.opcode$ is $=$. Also, when $e$ is the array name of an array reference, in the AST nodes $node$ and $node2 = ast(e)$, $node.kind$ is $ArraySubscriptExpr$, $node.inner[0]$ is $node2$, and $node2.kind$ is $DeclRefExpr$.

**For Q6 and Q7)** When $e$ is a library function, in $node = ast(e)$, $node.kind$ is $CallExpr$, and in the AST node $node2.id$ that matches $node.inner[0].inner[0].referencedDecl.id$, $node2.storageClass$ is $extern$. On the other hand, when $e$ is a user-defined function, $node2.storageClass$ is not $extern$. The $i$-th argument of $e$ is $node.inner[i]$.

## VI. Evaluation experiment

### A. Overview

The purpose of this experiment is to clarify the extent to which the proposal is effective in educational practice. To this end, we evaluate how accurately tracing tasks can be generated for the programs included in the textbook [8]. First, exercises using tracing tasks are conducted through interaction between the system and the learner. Therefore, we represent tracing tasks as a state transition diagram, where questions are states and transition conditions are the learner's answers. Then, the evaluator assesses the correctness and completeness of the questions and their generation conditions by comparing the C program with the state transition diagram.

### B. Evaluation method

We extracted three programs from each chapter of the textbook, from Chapter 1 to Chapter 9, and used them for evaluation after applying the following preprocessing steps. These programs include printf function, puts function, atoi function, user-defined functions, arithmetic operators, sizeof operator, ternary operator, assignment operator, arrays, if statements, for statements, while statements, do-while statements, and switch statements. Some programs with inappropriate line breaks and indentation are also included.

- We changed multibyte characters to single-byte characters. This is to facilitate the implementation of the state transition diagram generation tool described later.
- We changed the updating of variable values by the scanf function to updates by constants or variables. This removes the behavior of the scanf function from the learning scope. On the other hand, the scanf function makes the program's behavior dynamic, making it difficult to prepare feedback for tracing tasks in advance. In this research, we considered the latter disadvantage to be a significant issue.

The abstract syntax trees used for generating questions are generated using clang (version 14.0.0-1ubuntu1.1) on Ubuntu 22.04 LTS with the following command:

```
$ clang -Xclang -ast-dump=json path_to_source
```

The authors developed a tool to generate state transition diagrams based on Section V-B, using the abstract syntax tree generated by clang. The state transition diagram represents the interactive question generation by procedures P() and Q(e). In this context, the actions in question presentaion are represented as states (Table I), the selection of the next question independent of the answer (for example, the question about arguments in P2) is an unconditional transition, and the selection of the next question based on the answer (for example, P3 $\rightarrow$ P4) is a transition conditioned by the answer.

TABLE I
List of States

| State name | Action in question presentation | Related procedures |
|---|---|---|
| AssVal | Ask for the value to be assigned | Q1 |
| Calculate | Perform calculation | Q2, Q6 |
| CurVal | Ask for the current value | P2, Q4, Q5 |
| EvalVal | Ask for the evaluation result | P5 |
| Next caller | Ask for the return destination | P9 |
| Next expr | Ask for the next expression to be executed | P3 |
| Next func | Ask for the next user-defined function to be executed | P1 |
| Returned value | Ask for the return value | P6 |
| Skip | Do nothing | Q4, Q5, (Not applicable to any procedure) |
| UpdVal | Ask for the updated value | Q3 |

Figure 4 shows the state transition diagram generated from the C program shown in Figure 3. Ellipses represent states, and arrows represent transitions. The label of a state consists of the state name, identifier, and supplementary information, while the label of a transition represents the transition condition. For example, state A in the figure has the state name 'Next func', identifier 0, and supplementary information

'`1.1: void main(){...}`'. This means that it is the question about user-defined functions in P1 of P(), and its option is the main function starting from the 1st character of the 1st line. Transition B in the figure means that if the answer to P1 is the main function, it transitions to state C. State C in the figure has the state name 'Next expr', identifier 4, and supplementary information '`2.8: a = 0`', '`3.6: i = 1`', '`3.13: i < 3`', '`3.20: i++`', '`4.3: a = a + i`', '`5.2: printf("%d\n",a)`', and '`}`'. This means it is the question for P3, and its options are the seven aforementioned items.

On the other hand, the evaluator determines the system's behavior from the program in Figure 3. First, in P1, the system presents the main function as an option and asks about the user-defined function to be executed. If the learner answers with the main function, the system asks about the value of arguments in P2. However, since this main function has no arguments, the system does not implement this question. Next, in P3, the system presents '`a = 0`', '`i = 1`', '`i < 3`', '`i++`', '`a = a + i`', '`printf("%d\n",a)`', and '`}`' as options and asks about the expression to be executed. The evaluator considers these behaviors to match the questions and their presentation conditions derived from the state transition diagram.

### C. Evaluation results and discussion

In Table II, the column for chapter titles lists the chapters in order from Chapter 1, and the evaluation column shows the evaluator's assessment of the state transition diagrams generated for the programs. This table indicates that correct state transition diagrams were generated for 17 programs.

For programs list0321, list0401, and list0607, the generation tool did not generate the 'EvalVal' state for P5, but instead generated the 'Skip' state. For programs list0515, list0811, list0906, and list0913, it incorrectly generated the 'CurVal' state. This state was meant to ask about the starting address of array references, which is excluded from the learning objectives by Q5. Instead, the generation tool should have generated the 'CurVal' state to ask about the value of array references in Q4. For program list0701, the generation tool generated the 'UpdVal' state instead of the 'Skip' state. For program list0706, the generation tool did not generate the 'CurVal' state for Q5. For program list0806, the generation tool generated an incorrect 'Next expr' state. This state includes the initialization of global variables as an option for P3. This is likely because the generation tool recognized this global variable initialization as being included in a user-defined function. Although the initialization of global variables is not included in procedures P() and Q(e), it is a concept that beginners should learn, so we would like to modify P() and Q(e) to include it in tracing tasks.

These errors can be detected based on a basic interpretation of the programs. Therefore, it is highly likely that the information necessary for such interpretation is included in the abstract syntax tree, and it is believed that the generation tool can create correct state transition diagrams by additionally interpreting this information. Consequently, it can be said that a certain level of effectiveness of the proposed method has been confirmed.

TABLE II
EVALUATION RESULTS

| Chapter title | Program ID | Evaluation |
|---|---|---|
| Let's get familiar first | list0101 | Correct |
| | list0107 | Correct |
| | list0114 | Correct |
| Operations and types | list0201 | Correct |
| | list0206 | Correct |
| | list0212 | Correct |
| Branching of program flow | list0301 | Correct |
| | list0311 | Correct |
| | list0321 | Does not ask about the value of switch statement condition |
| Repetition of program flow | list0401 | Does not ask about the value of while statement condition |
| | list0412 | Correct |
| | list0423 | Correct |
| Arrays | list0501 | Correct |
| | list0509 | Correct |
| | list0515 | Does not correctly ask about values assigned to multidimensional arrays in declaration statements |
| Functions | list0601 | Correct |
| | list0607 | Does not ask about the value of while condition |
| | list0619 | Correct |
| Basic types | list0701 | Does not ignore constant macros |
| | list0706 | Does not correctly ask about the values of operands for ~ operator and ? operator |
| | list0713 | Correct |
| Let's try creating various programs | list0801 | Correct |
| | list0806 | Does not ignore global variable initialization |
| | list0811 | Does not correctly ask about values assigned to arrays in declaration statements |
| Basics of strings | list0901 | Correct |
| | list0906 | Does not correctly ask about values assigned to multidimensional arrays in declaration statements |
| | list0913 | Does not correctly ask about values assigned to multidimensional arrays in declaration statements |

```
void main(){
        int i,a = 0;
        for(i = 1; i < 3; i++)
                a = a + i;
        printf("%d\n",a);
}
```

Fig. 3. Example of C program

## VII. CONCLUSION

This paper proposed a method for automatically generating tracing tasks to realize a system that presents learners with tracing tasks with feedback. Specifically, we proposed a model that formalizes tracing tasks and a method to generate data from programs using abstract syntax trees to implement the model. Through experiments, we demonstrated that tracing

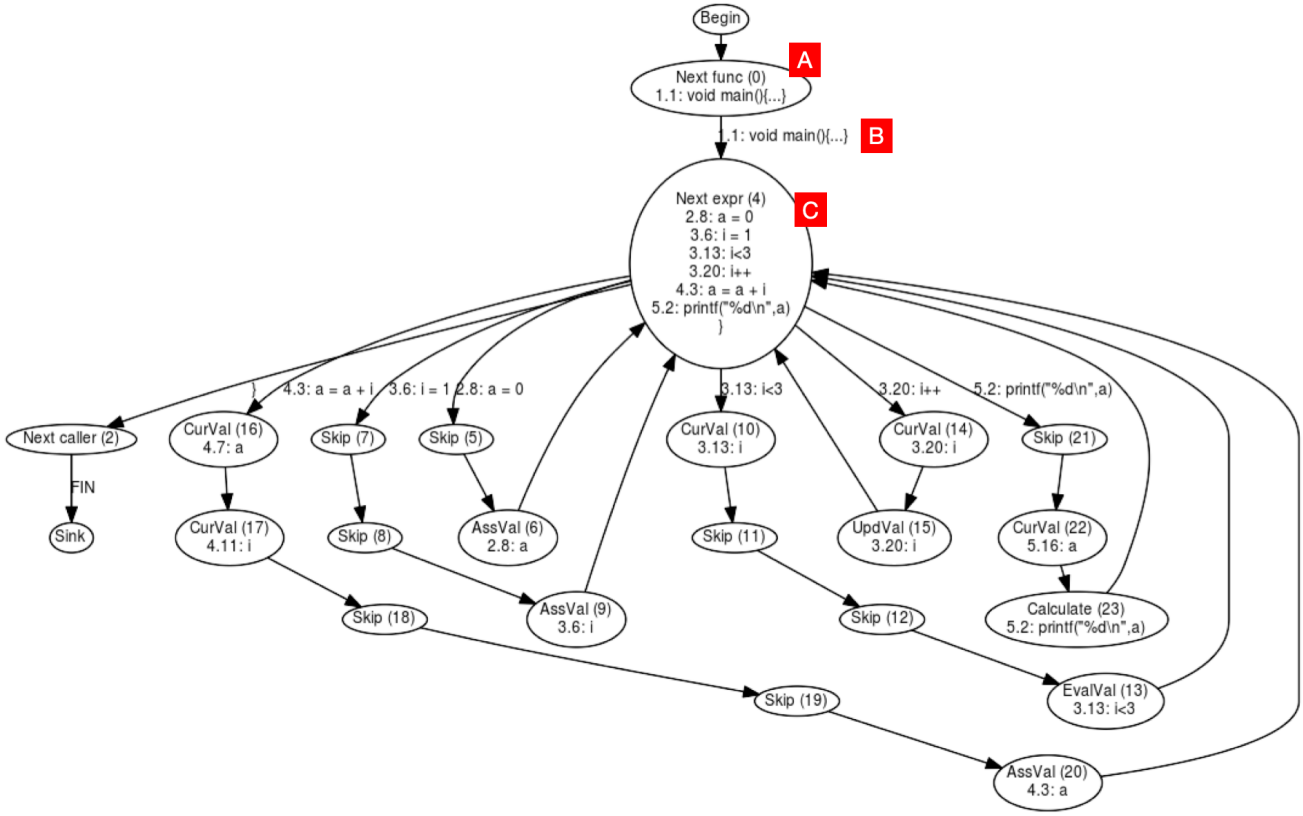


Fig. 4. Example of State Transition Diagram

tasks could be generated using the data extracted by the proposed method and the proposed model from programs in a programming exercise textbook. Through experiments, we demonstrated the effectiveness of the data extracted by the proposed method and the proposed model from programs in a programming exercise textbook for generating tracing tasks.

In future work, we plan to verify whether the proposed model and method can correctly generate tracing tasks from all programs in the textbook. Additionally, due to the variety of programming styles, we aim to further clarify the effectiveness of our proposal by expanding the scope to include programs from different textbooks and learners' programs. Subsequently, we will re-implement the tracing task generation of the system [2] using our proposal.

## ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Numbers JP22K12322, JP20H01730, and JP24K00454.

## REFERENCES

[1] T. Tomoto and T. Akakura, "Report on practice of a learning support system for reading program code exercise," in *Human Interface and the Management of Information: Supporting Learning, Decision-Making and Collaboration*, 2017, pp. 85-98.

[2] T. Mogi, Y. Tateiwa, T. Tomoto, and T. Akakura, "Evaluation of an Automatic Generation System for Tracing Tasks Based on Textbook Programs," in *Proc. 31st Int. Conf. Comput. Educ.*, 2023, pp. 327-332.

[3] K. Okimoto, S. Matsumoto, S. Yamagishi, and T. Kashima, "Developing a source code reading tutorial system and analyzing its learning log data with multiple classification analysis," in *Artif. Life Robotics*, vol. 22, 2017, pp. 227–237.

[4] K. Yamashita, T. Nagao, S. Kogure, Y. Noguchi, T. Konishi, and Y. Itoh, "Code-reading support environment visualizing three fields and educational practice to understand nested loops," in *Res. Pract. Technol. Enhanc. Learn.*, vol. 11, 2016, Art. no. 3.

[5] G. D. Blank, W. M. Pottenger, G. D. Kessler, S. Roy, D. R. Gevry, J. J. Heigl, S. A. Sahasrabudhe, and Q. Wang, "Design and evaluation of multimedia to teach java and object oriented software engineering," in *Proc. 2002 ASEE Annu. Conf. Expo.*, 2002, pp. 17–18.

[6] A. Omar, D. Barbara, F. Davide, G. Nicholas, and M. Alizadeh, "Learning recursion: Insights from the chiqat intelligent tutoring system," in *Proc. 12th Int. Conf. Comput. Supported Educ.*, vol. 2, 2020, pp. 336–343, SciTePress.

[7] GDB: The GNU Project Debugger. Accessed: Aug. 19, 2024. [Online]. Available: https://www.sourceware.org/gdb/

[8] B. Shibata, *Shin Meikai Cgengo Nyumonhen [New Lucidity C Programming Language Introductory]*, vol. 2, SB Creative, 2021.