Generation of Problems, Answers, Grade and Feedback – Case Study of a Fully Automated Tutor

Amruth N Kumar
Ramapo College of New Jersey
505 Ramapo Valley Road
Mahwah, NJ 07430-1680
amruth@ramapo.edu

ABSTRACT

Researchers and educators have been developing tutors to help students learn by solving problems. The tutors vary in their ability to generate problems, generate answers, grade student answers and provide feedback. At one end of the spectrum are tutors that depend on hand-coded problems, answers and feedback. These tutors can be expected to be pedagogically effective since all the problem-solving content is carefully hand-crafted by a teacher. However, their repertoire is limited. At the other end of the spectrum are tutors that can automatically generate problems, answers and feedback. They have an unlimited repertoire, but it is not clear that they are effective in helping students learn. Most extant tutors lie somewhere along this spectrum.

In this paper, we will examine the feasibility of developing a tutor that can automatically generate problems, generate answers, grade student answers and provide feedback. We will investigate whether such a tutor can help students learn. For our study, we considered a tutor for our Programming Languages course, which covers static and dynamic scope, i.e., static scope of variables and procedures, dynamic scope of variables, and static and dynamic referencing environment of procedures in the context of a language that permits nested procedure definitions. The tutor generates simple and complex problems on each of these five topics, solves the problems, grades the student's answers and provides feedback about incorrect and missed answers. Our evaluation over two semesters shows that the feedback provided by the tutor helps improve student learning.

1. INTRODUCTION

Several educators have used problem-solving as a means to incorporate active learning into Computer Science. In this approach, students solve well-structured problems, i.e., problems for which there are fixed, formulaic answers. Problem-solving is used to supplement classroom instruction. (This is in contrast with problem-based learning [5] where contextualized, ill-structured problems are used to *drive* the instruction.) Researchers and educators have been building software systems to facilitate problem-solving in Computer Science. These systems have been designed for different purposes:

- Some systems are designed for assessment/self-assessment. These systems grade the student's answers and let the student know whether or not the answers are correct (e.g., TuringsCraft [2], JFLAP [11]).
- Other systems are designed for learning/self-paced learning. These systems provide feedback in addition to grading the student's answer (e.g., WADEIn [10], CMeRun [13], Expresso [17]).

We will henceforth refer to these software systems as tutors.

Tutors may be categorized based on the nature of the problem-solving activity that they promote. Some of the stages in Bloom's taxonomy [7] targeted by the systems include:

- Knowledge, including recall of information, targeted by systems that administer multiple choice questions such as WebCT (www.webct.com).
- Application, including predicting the output of programs, targeted by tutors such as WADEIn [10] and the programming tutors that we have developed [12,21,32,33].
- Analysis, including debugging programs, targeted by tutors such as CMeRun [13], Expresso [17], and the programming tutors that we have developed [25,27].
- Synthesis, including writing programs, targeted by tutors such as TuringsCraft [2], LISP Tutor [31], PROUST [18], BRIDGE [8], and ELM-ART [34].

Tutors may be categorized into three types based on the problems they use:

1. Some tutors solve problems entered by the student, such as JFLAP [11] for Automata Theory. These tutors are the most flexible in terms of supporting the student's learning. They must be able to handle incorrectly formulated problems.
2. Some systems administer problems provided by the instructor, such as WebToTeach, now TuringsCraft [2]. A limited repertoire of problems is a drawback of these systems.
3. Some systems generate problems – examples include PILOT [9] for graph algorithms, and Gateway Labs for mathematical foundations of Computer Science [3]. Often, they generate problems as instances of parameterized templates [4,6,12,19,20,21,25,27,32,33]. This enables the tutors to present multiple un-identical instances of a problem, either to the same user on different occasions to provide for repetitive practice, or to different users on the same occasion to prevent plagiarism / cheating.

Can a tutor *automatically* generate problems that are correct according to domain principles? This would provide for repetitive practice while minimizing the possibility that a student may see the same problem or problem pattern twice. We will address this question in this paper.

Tutors use different mechanisms to obtain the correct answer and/or grade the student's answer:

- Some tutors compare the student's answer against the correct answer entered by the instructor. Such tutors have a limited repertoire of problems, and are more laborious to build since the correct answer for each problem must be encoded into the tutor.
- Some systems verify the correctness of the student's solution without the benefit of a correct solution (e.g., PILOT [9]).
- Some tutors run the student's solution through an independent problem solver such as a compiler to determine whether the answer is correct (e.g., TuringsCraft [2], InStep[30], CMERun [13], Expresso[17]).
- Some tutors implement an on-board problem-solver to solve the generated problems (e.g., WADEIn [10], JFLAP [11]). The problem-solver is restricted to the class of problems presented by the tutor, but is capable of explaining its solution.

In the context of a tutor that *automatically* generates problems, we would like to investigate whether such a tutor can also automatically solve the problems to generate the correct answer, and independently grade the student's answer without recourse to a grading schema.

Tutors differ as to when they provide feedback. Some systems provide immediate feedback [1], i.e., feedback while the student is attempting the problem. Examples include LISP Tutor [31], PROUST [18] and BRIDGE [8]. Others provide feedback only after the student has solved the problem, also referred to as demand feedback [1]. Examples include WADEIn [10], CMeRun [13], Expresso [17] as well as most of the tutors that we have developed for programming topics [12,21,25,27,32,33].

Tutors differ in terms of what they provide as feedback. Often, the feedback consists of pointing out the errors in the student's answer (e.g., TurinsgCraft [2], CMeRun [13], Expresso [17]). In addition, tutors may explain the correct answer [12,21,25,27,32,33]. Ideally, tutors must also explain why the student's answer is incorrect and/or how to fix it. We will investigate whether a tutor can automatically generate such feedback, and if so, whether the generated feedback is effective in helping students learn.

Tutors differ in how they obtain the feedback that they provide to the student. Tutors may simply display hand-crafted feedback incorporated into them by the instructor, indexed by problems and the types of errors in the student's answer. Alternatively, tutors may generate feedback by collating snippets of explanation based on the context, an approach we have used in our programming tutors [12,21,25,27,32,33]. Tutors may also generate the feedback automatically, tailored to the problem and the student's answer, which is the most flexible approach. We will consider this last approach in this paper.

To summarize, we will address the following questions in this paper: whether a problem-solving tutor can automatically generate problems, generate the correct answer to the problems, grade the student's answer, and generate feedback, and whether such a tutor is effective in helping students learn. We will examine these questions in the context of a tutor on static and dynamic scope, designed for students in the Programming Languages course. Figure 1 summarizes the tutor design options we have discussed so far. The tutor we will describe in this paper is close to the right end of the spectrum of choices, with automatic problem generation, on-board answer generation, automatic partial credit, dynamic feedback generation and the ability to explain why an answer is incorrect.

In Section 2, we will list the concepts addressed by the scope tutor. In Section 3, we will describe the automated generation of problems, answers, grade and feedback in the scope tutor. In Section 4, we will describe evaluation of the tutor in two semesters of our Programming Languages course – the protocol and the results. We will follow with a brief discussion of the results in Section 5.

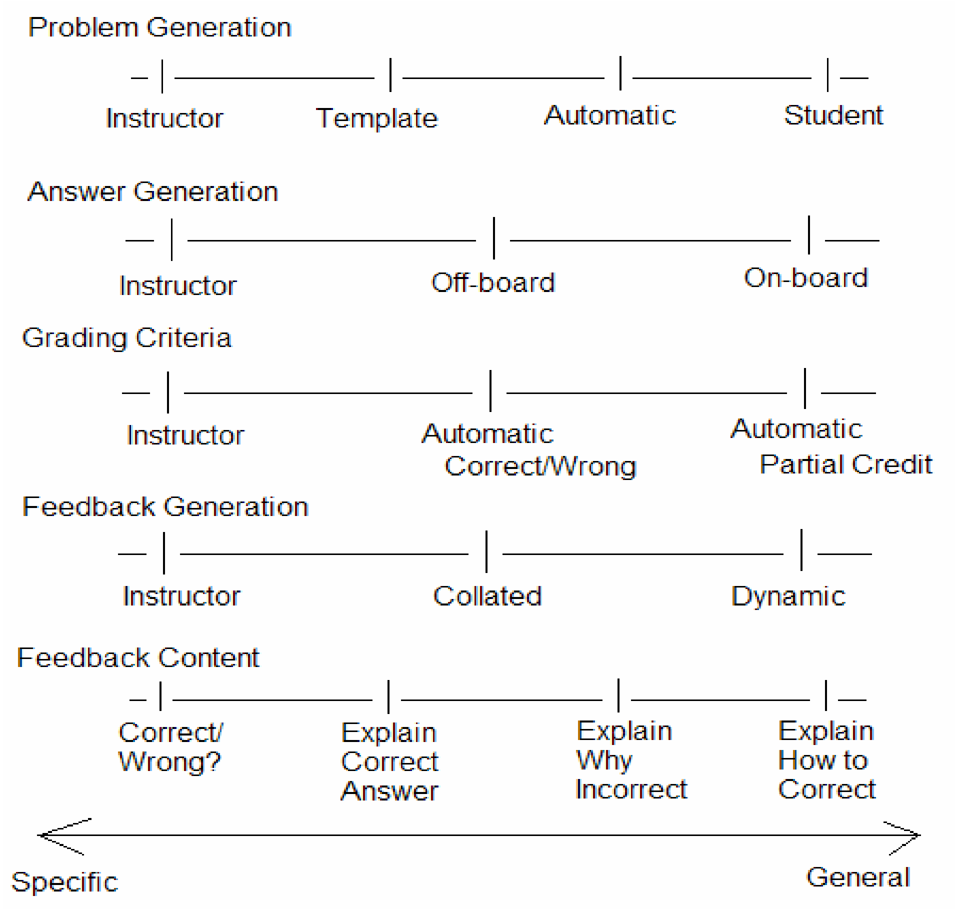


Figure 1: The spectrum of tutors.

2. THE DOMAIN

In the junior/senior-level Programming Languages course, static and dynamic scope are studied in the context of a language that permits nesting of procedure definitions, such as Pascal or Ada, because:

- The rules for determining the scope of a variable in such languages are a generalization of the rules in most other statically scoped languages (such as Java, C++, C#), including associated concepts such as namespaces, and nested classes.
- The rules for determining the scope of procedure names is significantly more complicated than in the other languages.

A study of static scope includes the following concepts:

1. Static Scope of variables. Static scope of a variable includes all the procedures in which the variable can be referenced. Static scope of a variable is defined as extending throughout the most immediately enclosing scope object, minus any enclosed scope object where the variable has been re-defined. For our purposes, scope objects are procedures.
2. Non-local Static Referencing Environment of procedures. Non-local variables of a procedure are variables that the procedure can reference, that have not been declared in the procedure. Non-local referencing environment of a procedure consists of all the non-local variables with unique names whose scope extends to the procedure. Typically, this is defined as all the unique variables declared in the ancestors of a procedure that have not been re-declared either in the procedure or in a nearer ancestor of the procedure.

3. Static scope of procedure names. Static scope of procedure names determines which procedure can call which other procedures in the hierarchical tree of procedure definitions. The converse of static scope of procedure names is procedure callability. In Pascal, procedure callability is defined as follows: A procedure can call itself, all the procedures defined directly inside it, all the ancestors of the procedure except main, and all the earlier siblings of these ancestors, i.e., sibling procedures that were defined before the ancestors in the program.

Static scope and referencing environment can be determined by studying the text of the program.

A study of dynamic scope includes the following concepts:

4. Dynamic Scope of variables. The dynamic scope of a variable is defined as extending to all the procedures called after the procedure in which it is declared, until and excluding the first procedure called that re-declares it.
5. Non-local Dynamic Referencing Environment of procedures. Once again, non-local variables of a procedure are variables that are not declared in the procedure that can still be referenced in it. Non-local dynamic referencing environment of a procedure consists of all the unique variables declared in the procedures called earlier in the procedure call sequence (called predecessors), that have not been re-declared in the procedure in question or its nearer predecessors.

Dynamic scope and referencing environment can be determined by studying the procedure call sequence.

Most students in the Programming Languages course are already familiar with static scope, but not in the context of a language that permits nested procedure definitions. Static scope of procedure names is hard for most students who are only familiar with C++, Java or C#. Dynamic scope is new to most students in the Programming Languages course since they are unlikely to have used a language with dynamic scope, such as APL, Snobol4, or early versions of LISP. Therefore, we developed a tutor on static and dynamic scope for use in the Programming Languages course. We used Pascal to illustrate static scoping concepts. In order to illustrate dynamic scope, we used a hypothetical language with Pascal syntax. Using the same syntax for both static and dynamic scope was meant to reduce the cognitive load on the students using our tutor.

3. THE TUTOR

Our scope tutor generates problems, grades the student's answer and provides feedback on each of these five topics. In the rest of this section, we will describe how the tutor automatically generates problems, the types of problems that it generates, the problem-solving support it provides, how the tutor automatically solves the generated problems, and how it automatically generates the feedback provided to the user.

3.1 Automatic Problem Generation

A tutor may dynamically generate problems in one of two ways:

1. Template-based generation: The tutor generates problems as random instances of parameterized problem templates that are built into the tutor by the instructor. E.g., a template for selection statements might be:
   ```
   if( <V1> > <V2> ) print <V1>; else print <V2>;
   ```
   In the template, <V1> and <V2> are parameters. The tutor generates the code corresponding to this template by randomly selecting variables for the parameters <V1> and <V2>. The resulting code may be:
   ```
   if( index > count )
       print index;
   else
       print count;
   ```
   Since <V1> and <V2> are randomly chosen each time a problem is generated from this template, no two problems generated from the template are identical. This is the mechanism that we have used in most of our programming tutors [12,21,25,27,32]. Other researchers have also used this mechanism for problem generation, e.g., [9]. An advantage of this approach is that the form of the problem and its level of difficulty can be closely controlled. A disadvantage is that the tutor designer must enter a repository of problem templates into each tutor, which is a labor-intensive and error-prone task.
2. Automatic generation: The tutor generates problems based on a "model" of the problem domain. The tutor knows how to generate problems based on the domain model, how to solve them and how to provide feedback for the problems. This is truly dynamic and random generation of problems in that not even the tutor

designer knows what problem the tutor might generate next. This is the approach we have taken for our Scope tutor as well as a few earlier tutors on expression evaluation and selection statements [14,15,22,28,29,33].

In our experience, template-based generation of problems is more appropriate for problems on the semantics of programs, and automatic generation of problems is more appropriate for problems on the syntax of programs.

The nesting relationship of procedures in a Pascal program can be illustrated as a tree, with the main program as the root and the nested procedures as intermediate and leaf nodes. This is called the static tree of the program. The static tree is one of the two domain "models" used by our tutor to generate problems. The tutor uses the following overall algorithm:

```
While the student has not quit:
1. Generate a new static tree, and the outline of a Pascal program corresponding to it.
2. While additional problems can be posed based on the program, or a pre-set limit has not been
   reached for the number of problems based on the program:
     a. Generate a problem, present it to the student and obtain the student's answer;
     b. Solve the problem and generate the correct solution;
     c. Compare the student's answer against the correct answer to determine the student's grade
        and feedback, and present them.
     d. Update the student's grade.
```

Generation of Static Tree: The tutor dynamically generates the static tree. It determines the following parameters randomly:

1. The number of levels of nesting in the tree.
2. The number of procedures at each level in the tree.
3. The number of variables declared in each procedure.

We use the following heuristic definition of hardness of problems, designed to minimize the probability of a student arriving at the correct answer through random guesses or the simple process of elimination:

1. The more the answering options for a problem, the harder the problem. For instance, the static scope of a variable declared in the root of a tree can potentially extend to more procedures in a tree of depth 4 than in a tree of depth 2.
2. The more the number of components in the correct answer, the harder the problem. E.g., a problem on static referencing environment whose answer includes 5 variables is harder than a problem whose answer includes only 2 variables. Of course, the number of components in the correct answer must be less than the number of answering options for the problem.

All the parameters of the static tree that we listed above affect the hardness of the problems that can be generated from the static tree. The static tree may contain up to 20 procedures and 2-5 levels of nesting.

The tutor also randomly determines the names of the procedures and the names of the variables in those procedures. This is cosmetic randomization that does not affect the semantics of the static tree or the hardness of the problems generated from it. However, it adds to the variability of the problems. Therefore, the tutor seldom generates the same static tree twice.

The tutor translates the static tree into the outline of a Pascal program, which serves as the stem for the generated problems. It uses the information about the nesting of procedure definitions, names of procedures and names of variables from the static tree to generate the program text. (See Figure 2). The generated program is an outline – it contains only the code needed to resolve scope issues. The program does not purport to have any execution semantics. Providing an outline rather than a complete program is a form of scaffolding that helps focus the attention of the student during problem-solving. It serves to highlight the parts of a typical program on which the student should focus when analyzing scope in a complete program later on.

The other domain model used by the tutor is the procedure call sequence. This is the sequence in which procedures are called when the program is executed. The tutor dynamically generates a sequence of procedure calls by randomly picking procedure names, subject to the rules of procedure callability. The tutor represents the resulting procedure call sequence as a chain.

3.2 Types of Problems

From each static tree, the tutor can generate numerous problems on each of the five topics. The tutor can generate problems at two different levels: simple problems which involve just one variable/procedure, and complex problems which involve two variables/procedures. The simple and complex problems generated by the tutor on each of the five topics are:

- **Static Scope:**
    - Simple Problem: Select all the procedures which lie within the scope of variable *v* declared in procedure *p* (See Figure 3).
    - Complex Problem: Select all the procedures which lie within the scope of both the variable *v1* declared in procedure *p1* and the variable *v2* declared in procedure *p2*.
- **Static Referencing Environment:**
    - Simple Problem: Select all the variables which lie within the non-local referencing environment of procedure *p*.
    - Complex Problem: Select all the variables which lie within the non-local referencing environment of both procedure *p1* and procedure *p2*. Do not include local variables in either procedure.
- **Procedure Callability:**
    - Simple Problem: Select all the procedures which the procedure *p* can call.
    - Complex Problems:
        - Select all the procedures which the procedure *p1* can call but the procedure *p2* cannot call (See Figure 4).
        - Select all the procedures that both procedure *p1* and procedure *p2* can call.
- **Dynamic Scope:**
    - Simple Problem: Given a sequence of procedure calls, select all the procedures which lie within the scope of the variable *v* declared in the procedure *p* (See Figure 5).
    - Complex Problem: Given a sequence of procedure calls, select all the procedures which lie within the scope of all the variables declared in the procedure *p*.
- **Dynamic Referencing Environment:**
    - Simple Problem: Given a sequence of procedure calls, select the variables in the non-local referencing environment of procedure *p1*.
    - Complex Problem: Given a sequence of procedure calls, select the variables that lie in the non-local referencing environment of both procedure *p1* and procedure *p2* (See Figure 6).

In order to generate a simple problem on a topic, the tutor randomly picks a variable and/or procedure from either the static tree (for static scope) or the procedure call sequence (for dynamic scope). If the static tree contains *p* procedures and each procedure contains an average of *v* variables, the tutor can generate *p*v* unique problems on static scope, *p - 1* problems on static referencing environment (excluding main), and *p* problems on procedure callability. With a procedure call sequence that involves all *p* procedures, it can again generate *p * v* unique problems on dynamic scope and *p – 1* problems on dynamic referencing environment (excluding main). In reality, the tutor changes the static tree after a fixed number of problems. The student can also request a new program, which results in the generation of a new static tree.

In order to generate complex problems, the tutor uses a generate-and-test mechanism: It randomly selects two variables/procedures, and solves the complex problem resulting from the combination of the variables/procedures to check if a valid answer exists. It repeats this process until it finds a combination of two variables/procedures that results in a problem with a valid answer.

The tutor can be configured to generate problems at three levels of difficulty: basic, intermediate and advanced. At the basic level, it generates simple problems only. At the intermediate level, it generates a mix of simple and complex problems, and at the advanced level, it generates complex problems only.

3.3 Problem-Solving Support

The outline of a Pascal program can be hard to read at a glance, especially when it involves several levels of nesting. To help students analyze the code, the tutor provides several formatting options:

- Color-Coding: The program can be displayed with a different color for each level of nesting. This helps the student quickly identify all the procedures at a given level of nesting.

- Boxes: The program can be displayed with a bounding box around each procedure (See Figure 3). This is especially useful for the students who are not very familiar with the Pascal syntax.
- Indentation: The level of indentation of the program can be changed.

All these mechanisms are designed to help the student identify the static tree corresponding to the program.

The tutor provides scaffolding in the form of two types of graphical representations of the program:

- The student can view the static tree of the program, displayed with procedure and variable names listed at each node (See Figure 4). This is helpful for answering problems on static scope.
- The student can view the procedure call sequence as a graphic chain, with procedure and variable names listed at each node (See Figure 6). This is helpful for answering problems on dynamic scope.

The graphical representations eliminate the clutter of program syntax and help the student focus on only what is essential to solve the problems. Initially, the tutor displays the outline of the Pascal program. The student can switch to the static tree or the procedure call sequence by selecting it from the View menu.

For problems on the static scope of variables, the static scope of procedure names and the dynamic scope of variables, the student must identify appropriate procedures. For these problems, the student is presented with a check-list of all the procedures defined in the program (See Figure 3). For problems on the static and dynamic referencing environment of procedures, the student must identify variables. Since a variable may have been declared in multiple procedures, the student must not only identify each variable, but also the procedure where the variable is declared. Therefore, a table is provided for the student to enter the answer, wherein rows are labeled with variable names and columns are labeled with procedure names (See Figure 6).

3.4 Automatically Solving the Generated Problems

In order to grade the student's answer, the tutor must first solve the problems to generate the correct answer. The tutor uses tree traversal algorithms to solve the generated problems. We will discuss these algorithms in this section.

The tutor determines the static scope of a variable *v* declared in a procedure *p* by performing a depth-first search of the sub-tree of the static tree that has the procedure *p* as the root node. The algorithm is as follows:

1. Start from the node for procedure *p* in the static tree. Add *p* to the scope of the variable *v*.
2. If *p* does not have any children, i.e., nested procedure definitions, return.
3. Else, visit each child *c* of *p*:
   - o If procedure *c* re-declares the variable *v*, return.
   - o Else, add procedure *c* to the scope of the variable *v*. Recursively execute step 2 onwards with procedure *c* instead of *p*.

The tutor determines the dynamic scope of a variable using the same algorithm, but applies it to the procedure call sequence instead of the static tree.

The tutor determines the static non-local referencing environment of a procedure *p* using the following algorithm:

1. Start from the node for procedure *p* in the static tree. Include all the variables in *p* to the set *S* of variables already seen.
2. Visit the parent *r* of procedure *p*. This is the procedure in which procedure *p*'s definition is nested. If *p* has no parent, i.e., *p* is the root of the static tree, return.
3. Add all the variables in *r* that are not also in *S* to the non-local referencing environment of *p*, as well as to the set *S*.
4. Recursively execute step 2 onwards with procedure *r* instead of *p*.

The tutor determines the dynamic non-local referencing environment of a procedure using the same algorithm, but applies it to the procedure call sequence instead of the static tree.

The tutor determines the procedures that a procedure *p* can call by using the following rules of tree traversal:

1. *p* can call any procedure that can be reached by following 0 or more links up the static tree. The root, i.e., `main` is an exception to this rule – no procedure can call main. This rule implies that *p* can call itself recursively.
2. For each procedure *a* reached in Step 1, *p* can call any procedure that can be reached from *a* by following 1 link down the static tree, but to the left of the upward link followed to reach *a*. This rule implies

that a procedure can call its own children. It can call the siblings of any of its ancestors that were defined before its ancestor in the program code.

For complex problems, the tutor solves the problem for each of the two variables/procedures to obtain two answer sets, and applies set operations (usually intersection or set difference) to these two answer sets to obtain the final answer.

3.5 Automatically Generating Feedback

The tutor provides feedback at two levels:

- **Error Flagging:** When the student selects an answer, the tutor highlights the answer in green if the answer is correct, and red if it is incorrect (See Figure 5). The student is allowed to unselect an answer and retry other answers. During this process, the tutor does not provide any explanation.
- **Demand Feedback:** Once the student submits the answer, the tutor provides demand feedback (See Figure 6). We will describe demand feedback in the rest of this section.

In addition, the tutor can be configured to provide no feedback, or minimal feedback wherein, it indicates whether an answer is correct or wrong, but does not explain why.

The answers to the problems are sets of procedures or variables - clearly, the order in which the elements of the set are identified by the student is unimportant. Given that the answers are sets, we propose a simple calculus for the types of feedback that the tutor must provide:
Assume that the actual answer is the set A and the student's answer is the set S:

- C = A n S, C being the set of correct answers;
- I = S – A, I being the set of incorrect answers;
- M = A – S, M being the set of missed answers.

These relationships are illustrated in Figure 7 below. The tutor must provide feedback for incorrect answers – why they are incorrect; and missed answers – why they are correct.

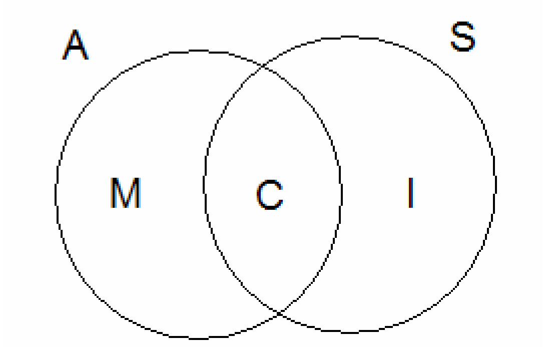


Figure 7: Relationship between the student's answer (S) and actual answer (A).

Some sample explanations provided by the tutor for incorrect and missed answers follow:

- **Static Scope:**
  - For each **missed** procedure, the tutor points out that the variable in question was not re-declared in the procedure or any of its intervening ancestor procedures. In order to do so, the tutor traverses the tree from the missed procedure up to the procedure where the variable was declared and enumerates all the intervening procedures in which the variable was not re-declared.
  - For each **incorrect** procedure, the tutor points out either that the procedure is not a descendant of the procedure that declared the variable, or that it or one of its ancestors re-declared the variable. In the latter case, the tutor traverses the tree from the procedure that declared the variable down to the incorrect procedure in order to locate and report the first intervening ancestral procedure that re-declared the variable.
- **Static Referencing Environment:**
  - For each **missed** variable, the tutor points out that the variable was declared in an ancestor, and was not re-declared in any intervening ancestral procedure. The tutor traverses the tree from the procedure in question up to the procedure where the missed variable was declared, and enumerates all the intervening procedures where the variable was not re-declared.

    - For each **incorrect** variable, the tutor points out either that the procedure that declares the variable is not an ancestor, or that the variable is re-declared in an intervening ancestral procedure, shadowing the visibility of the incorrectly selected variable. In the latter case, the tutor traverses the tree from the procedure that declared the incorrect variable down to the procedure in question in order to locate and report the first intervening ancestral procedure where the variable was re-declared.
- **Procedure Callability:**
    - For each **missed** procedure, the tutor points out why the procedure can be called: because it is a child, earlier sibling, ancestor, or an earlier sibling of an ancestor of the calling procedure. The tutor establishes the relationship between the calling procedure and the missed procedure by traversing the static tree.
    - For each **incorrect** procedure, the tutor points out why it cannot be called – because it is none of the above.
- **Dynamic Scope:**
    - For each **missed** procedure, the tutor points out how the variable was not re-declared in any procedure called between the procedure where the variable was declared and the missed procedure (inclusive). In order to do so, the tutor traverses the procedure call chain from the missed procedure up to the procedure where the variable was declared and enumerates all the intervening procedures in which the variable was not re-declared.
    - For each **incorrect** procedure, assuming that the incorrect procedure was called after the procedure in which the variable was declared, the tutor points out the first intervening procedure that re-declared the variable. In order to do so, the tutor traverses the procedure call chain from the procedure that declared the variable down to the incorrect procedure in order to locate and report the first intervening predecessor procedure that re-declared the variable.
- **Dynamic Referencing Environment:**
    - For each **missed** variable, the tutor points out how the variable was declared in a procedure in the call sequence and was not re-declared thereafter. The tutor traverses up the procedure call chain from the procedure in question up to the procedure where the missed variable was declared, and enumerates all the intervening procedures where the variable was not re-declared.
    - For each **incorrect** variable, the tutor notes either that the variable is not declared in any procedure called earlier in the call sequence, or that it has been re-declared in another earlier procedure in the call sequence. In the latter case, the tutor traverses the procedure call chain from the procedure where the incorrect variable was declared down to the procedure in question in order to locate and report the first intervening predecessor procedure where the variable was re-declared.

The tutor uses a simple overlay student model. It records the number of problems that the student generates, attempts, solves correctly, solves incorrectly, and solves partially for each topic. It presents this data to the student in a separate window. It combines the data for the various topics to provide an aggregate score. It also displays the level of hardness of the problem (basic / intermediate / advanced) last attempted by the student in each topic.

3.6 Graphical User Interface

The tutor uses the same graphical user interface for all five topics: a left panel for the program and its visualization, and a right panel for the problem, answering options and feedback provided to the student. This uniformity of interface helps reduce the learning curve for using the tutor. The student is led through an intuitive clockwise flow of action (as shown by the superimposed arrows in Figure 8): from the program to the problem statement (A), answering options (B), grading button (C), feedback (D), and the button to generate another problem (E) (See Figure 8). The tutor provides menu options to select the hardness level of the code (easy / normal), the hardness level of the problem (basic / intermediate / advanced), the type of feedback provided (none / minimal / error flagging / detailed), how frequently a new static tree must be generated (for every problem / every 5 problems / every 10 problems), and to generate a new program / problem.

4. EVALUATION OF THE TUTOR

We have presented a tutor on scope that does all of the following automatically: generate problems, solve them, grade the student's answer and generate feedback. We wanted to evaluate whether using such a tutor would help students learn. In particular, *could students learn from the feedback automatically generated by the tutor?*

Recall that when configured to provide "minimal feedback", the tutor indicates whether an answer is correct or wrong, without explaining why. In the following discussion, in addition to indicating whether an answer is correct or not, when the tutor includes explanation of incorrect and missed answers, we will refer to it as "detailed feedback".

4.1 The Protocol

We evaluated the tutor in fall 2001 and fall 2002. We used a within-subjects design. The protocol consisted of three stages:

1. **No-Practice Stage:** We conducted this stage to evaluate practice effect and/or Hawthorne effect [16]. We used problems on *static scope and referencing environment* for this stage. The stage consisted of two steps:
    a. Pre-test: Students took a written pre-test for 8 minutes. The test consisted of 4 programs, with 5 problems per program.
    b. Post-Test: The students took a written post-test for 8 minutes. The post-test also consisted of 4 programs, with 5 problems per program.

    Since the students did not get any practice between the pre-test and the post-test, any increase in the score from the pre-test to the post-test can be attributed to increased familiarity of the students with the types of the problems on the tests, brought on by repeated practice.
2. **Minimal Feedback Stage:** We conducted this stage to evaluate the effect of minimal feedback. We used problems on *dynamic scope and referencing environment* for this stage. The stage consisted of three steps:
    a. Pre-test: Students took a written pre-test for 8 minutes. Once again, the test consisted of 4 programs, with 5 problems per program.
    b. Practice: Students practiced with the tutor for 12 minutes. During the practice, the tutor provided minimal feedback – it listed whether the student's answer was correct or wrong, but not why.
    c. Post-test: Students took a written post-test for 8 minutes. The post-test was structured similar to the pre-test.
3. **Detailed Feedback Stage:** We conducted this stage to evaluate the effect of detailed feedback. We used problems on *procedure callability* for this stage. The stage again consisted of three steps:
    a. Pre-test: Students took a written pre-test for 8 minutes. The test consisted of 8 programs, with 5 problems per program.
    b. Practice: Students practiced with the tutor for 12 minutes. During the practice, the tutor provided detailed feedback – it not only listed whether the student's answer was correct or wrong, but also provided explanation for incorrect and missed answers.
    c. Post-test: Students took a written post-test for 8 minutes. The post-test was structured similar to the pre-test.

These stages and their constituent steps were conducted back-to-back, with no breaks in between. Students did not have access to their textbook, notes, or any other reference material during this evaluation. They were not allowed to consult each other or ask questions during the evaluation. Therefore, any change of score from pretest to post-test during the second and third stages was attributable solely to the use of the tutor for practice.

This evaluation was conducted as part of the Programming Languages course. It was conducted in a laboratory after the corresponding material had been covered in class. The author was the instructor of the course. He administered the evaluation and was present in the laboratory during the duration of the evaluation. The students worked on individual computers. They did not have access to the tutor before the evaluation.

The various written pre-tests and post-tests were developed by the author. The programs and problems on the tests were culled from previous mid-term exams in the course. The problems were similar to the problems presented by the tutor. Answers to the problems were of fixed-form (variable/procedure names) and therefore, the grading was objective. Students were told that the higher of the pre-test and post-test scores from each stage would be credited towards their course grade. This provided incentive for the students to do their best on *all* the tests during the evaluation.

4.2 The Results

We have listed the average score, and the average number of problems solved on the pre-test and the post-test during the three stages of evaluation in Tables 1 (Fall 2001) and 2 (Fall 2002). Note that in the first stage, the scores increased by 50% and the number of problems solved by the students increased by 30-42% from the pre-test to the post-test. The increase in the scores is statistically significant in both the semesters. Since we did not provide any practice between the pre-test and the post-test during the first stage, we can only attribute this increase to the increased familiarity of the students with the types of problems included in the tests. In order to eliminate this practice effect, we will henceforth consider the score per attempted problem rather than the raw score in our analysis.

Table 1: Scores and Problems Solved in Fall 2001

Fall 2001 (N=7)	**Pre-Test**		**Post-Test**		**% Change**		**p-value**	
	Score	Problems	Score	Problems	Score	Problems	Score	Problems
No Practice (Static Scope)								
Average	31.00	10.00	46.43	14.29	49.77%	42.86%	0.0002	0.011
Std-Dev	18.77	3.32	17.89	3.15				
Minimal Feedback (Dynamic Scope)								
Average	29.86	11.57	36.71	15.14	22.97%	30.86%	0.280	0.118
Std-Dev	28.15	4.72	27.78	8.17				
Detailed Feedback (Procedure Call)								
Average	30.57	16.29	66.29	22.14	116.82%	35.96%	0.009	0.043
Std-Dev	18.05	4.31	26.99	6.23				

Table 2: Scores and Problems Solved in Fall 2002

Fall 2002	**Pre-Test**		**Post-Test**		**% Change**		**p-value**	
	Score	Problems	Score	Problems	Score	Problems	Score	Problems
No Practice (Static Scope) (N=6)								
Average	24.83	7.83	37.33	10.17	50.34%	29.79%	0.006	0.519
Std-Dev	10.98	2.48	8.41	2.48				
Minimal Feedback (Dynamic Scope) (N=7)								
Average	35.86	11.00	41.57	16.00	15.94%	45.45%	0.074	0.013
Std-Dev	20.40	4.55	16.25	6.06				
Detailed Feedback (Procedure Call) (N=7)								
Average	35.43	16.29	79.00	23.29	122.98%	42.98%	0.00005	0.017
Std-Dev	18.87	5.02	26.03	8.36				

We have listed the score per attempted problem from fall 2001 in Table 3 and fall 2002 in Table 4. In each table, we have listed the class average and standard deviation of the score per attempted problem on the pre-test and the post-test, followed by the percentage change in the average score, and the corresponding t-test 2-tailed p-value.

Table 3: Score per Problem in Fall 2001

Fall 2001 (N = 7)	**Pre-Test Score / Problem**	**Post-Test Score / Problem**	**% Change**	*t-test* ***p*-value**
No Practice (Static Scope)				
Average	2.85	3.27	14.83%	0.17067
Std-Dev	1.57	0.99		
Minimal Feedback (Dynamic Scope)				
Average	2.35	2.24	-4.76%	0.80252
Std-Dev	1.61	0.86		
Detailed Feedback (Procedure Call)				
Average	1.95	2.93	50.46%	0.03071
Std-Dev	1.04	0.73		

Table 4: Score per Problem in Fall 2002

Fall 2002	**Pre-Test Score / Problem**	**Post-Test Score / Problem**	**% Change**	*t-test* ***p*-value**
No Practice (Static Scope) (N=6)				
Average	3.11	3.71	19.36%	0.08093
Std-Dev	0.65	0.51		
Minimal Feedback (Dynamic Scope) (N=7)				
Average	3.10	2.69	-13.26%	0.18565
Std-Dev	0.71	0.59		
Detailed Feedback (Procedure Call) (N=7)				
Average	2.17	3.45	58.76%	0.01102
Std-Dev	0.82	0.26		

The results from the two semesters are consistent with each other:

- When no practice was provided between the pre-test and the post-test, there was an improvement of 15-19% from the pre-test to the post-test, although this improvement was not statistically significant. As mentioned earlier, this improvement can be attributed purely to increased familiarity of the students with the types of problems included in the tests.
- When the tutor provided minimal feedback, as is typically done in textbooks, there was no statistically significant change from the pre-test to the post-test. In the absence of any new information that can support student learning, merely repeatedly solving problems cannot be expected to improve learning. Therefore, minimal feedback does not promote learning.
- When the tutor provided detailed feedback, the student scores improved by 50-58%, and this improvement was statistically significant (p-value < 0.05). Since the students did not have access to any other instructional materials during this stage, this improvement can be attributed to the practice provided by the tutor between the two tests. Since this tutor differed from the tutor in the previous stage in terms of the amount of feedback it provided to the student, i.e., minimal versus detailed feedback, the improvement from the pre-test to the post-test can be attributed to the detailed feedback provided by the tutor during practice. In other words, the feedback automatically generated by our tutor does help students learn.

One weakness of our study is our sample size: it is small. This weakens the lack of statistical significance of the results for no practice and minimal feedback cases, but *strengthens* the significance of the improvement with detailed feedback. Moreover, we used a within-subjects design, the improvement from pre-test to post-test was statistically significant only when the tutor provided detailed feedback and this result was consistent across two semesters, all factors contributing to our conclusion.

Affective Feedback: Students filled out feedback forms after the second and third stage in the protocol. The feedback forms contained 12 questions on the usability, learnability and usefulness of the tutor. Students responded to each question on a Likert scale of 1 (Strongly Agree) through 5 (Strongly Disagree). For analysis purposes, we combined the data from fall 2001 and fall 2002. Since we used a within-subjects design, the feedback forms from the two stages were filled by the same group of students, but in response to different treatments: minimal versus detailed feedback. The difference in the feedback provided after the minimal and detailed feedback stages was particularly telling on the following three questions:

Table 5: Affective Feedback Results from Fall 2001 and Fall 2002 Combined

Question	Average Response		*t-test* *p*-value
	Minimal	Detailed	
Feedback was sufficient	2.92	1.69	0.0129
Tutor helped me better understand what I knew	2.50	1.29	0.0056
Tutor helped me learn new material	3.27	1.93	0.0025

Note that all three differences are statistically significant. Clearly, students considered detailed feedback more adequate than minimal feedback. They also thought that the tutor that provided detailed feedback helped them learn better than the tutor that provided minimal feedback.

After the minimal feedback stage, a student wrote the following on the feedback form:

- “I got confused so the feedback would have been very welcome. :)”

After the detailed feedback stage, students wrote the following comments on the feedback form, indicating that they preferred detailed feedback:

- "I immediately caught on to the idea here. Excellent tutor."
- "I like the feedback because it helps me see what I did wrong. Lots of examples help me understand things easier and this helped show me what I was doing wrong."

5. DISCUSSION

We described a tutor on scope in programming languages, that generates all of the following automatically: the problems, the correct answer, and the feedback. Given this case study, let us revisit the questions we asked early in this paper:

- Can a tutor *automatically* generate problems? Yes, if it has the right model. In our case, the model was the static tree of a program for static scope, and the procedure call sequence for dynamic scope. In the past, we have used a tree to automatically generate problems on expression evaluation [22] and nested selection statements [33]. The model does not have to be derived from the domain, as long as it captures all the essential domain concepts.
- Can a tutor *automatically* solve the generated problems? Yes. We have used simple tree traversal algorithms in our scope tutor to automatically solve the generated problems. For our more recent work on programming tutors [12,21,25,27,32,33], we have been using domain models which are capable of automatically solving any problem in the domain [24,26]. These domain models are derived from the domain unlike the model used in this tutor.
- Can a tutor *automatically* grade the student's answer without recourse to a grading schema entered by the instructor? Yes. We have proposed a simple calculus for grading and the types of feedback that can be provided when the correct answer is a set.
- Can a tutor *automatically* generate feedback? Yes, as long as the model used for problem generation captures all the relevant domain concepts. We have used simple tree traversal algorithms in our scope tutor to automatically generate feedback for missed and incorrect answers.
- Can automatically generated feedback be effective? Yes, as demonstrated by the results of our evaluation.

A problem-solving tutor must incorporate an appropriate model in order to be able to automatically generate problems, answers and feedback. From our experience, we have identified two types of programming tutors based on their model:

1. Tutors on the execution (semantics) of the program. These need a language interpreter to solve the problem. Examples include tutors on pointers [25,27], expression evaluation [23], loops [12], classes [21] and parameter passing in programming languages [32]. These models are not domain-independent, and are hence, not reusable. However, they are composable, e.g., the model of a tutor on loops can be combined with the model of a tutor on selection statements to generate a tutor on loops that include selection statements.
2. Tutors on the structure (e.g., syntax or static-semantics) of the program. These do not need a language interpreter – they can use any model that captures the domain concepts, whether or not the model is derived from the domain language. Scope tutor [15,28,29] belongs to this category. Type compatibility, function overloading and polymorphism are examples of other topics for which such tutors can be built. The domain-independent models may be reusable for isomorphic concepts, but they are not composable.

We have since extended our scope tutor to include implementation techniques for static and dynamic scope. These include static chaining and display method for static scope and shallow and deep access for dynamic scope [14]. With these extensions, the tutor covers all the aspects of scope typically covered in a Programming Languages course.

Currently, we are working on developing a comprehensive suite of tutors for the introductory Computer Science course. These tutors use a custom-built language interpreter as their model. They generate problems based on templates. The generation of problems is adaptive, i.e., the tutors generate problems tailored to the learning needs of the student. The tutors are capable of automatically generating explanation of the execution of programs, and this explanation is included in the feedback. Some of the topics for which we have developed tutors include arithmetic and relational expression evaluation, selection statements, counter-controlled and pre-test logic-controlled loops, pointers and classes. Results of evaluation of the tutors indicate that the tutors help students learn programming concepts; and that the tutors are more effective when they provide detailed feedback than when they provide minimal feedback.

5. ACKNOWLEDGMENTS

A significant portion of the tutor was implemented by Eric Fernandes. Partial support for this work was provided by the National Science Foundation's Course, Curriculum and Laboratory Improvement Program under grant DUE-0088864, and Educational Innovation Program under grant CNS-0426021.

6. REFERENCES

[1] Anderson J.R., Corbett A.T., Koedinger K.R. and Pelletier R. 1995. Cognitive Tutors: Lessons Learned. *The Journal of the Learning Sciences* 4(2), 167-207.

[2] Arnow, D. and Barshay, O. 1999. WebToTeach: An Interactive Focused Programming Exercise System. In *Proceedings of Frontiers in Education Conference*, San Juan, Puerto Rico, November 1999, Session 12a9.

[3] Baldwin, D. 1996. Three years experience with Gateway Labs. In *Proceedings of Innovation and Technology in Computer Science Education Conference,* Barcelona, Spain, June 1996, ACM Press, 6-7.

[4] Barker, D.S., CHARLIE. 1997. A Computer-Managed Homework, Assignment and Response, Learning and Instruction Environment. In *Proceedings of Frontiers in Education Conference*, Pittsburgh, PA, November 1997, Session S4D.

[5] Barrows, H.S. 1986. A Taxonomy of Problem-Based Learning Methods. In *Medical Education*, 20, 481-486.

[6] Belmont, M.V., Guzman, E., Mandow, L., Millan, E., and Perez-de-la-Cruz, J.I. 2002. Automatic generation of problems in web-based tutors. In *Virtual Environments for Teaching & Learning*, L.C. Jain, R.J. Howlett, N.S. Ichalkaranje and G. Tonfoni (ed.), World Scientific, 2002, 237-282.

[7] Bloom, B.S. and Krathwohl, D.R. 1956. *Taxonomy of Educational Objectives: The Classification of Educational Goals, by a committee of college and university examiners. Handbook I: Cognitive Domain*. New York, Longmans, Green. 1956.

[8] Bonar, J. and Cunningham, R. 1988. BRIDGE: Tutoring the programming process. In *Intelligent tutoring systems: Lessons learned*. J. Psotka, L. Massey, and S. Mutter, Eds. Lawrence Erlbaum Associates, Hillsdale, NJ, 409-434.

[9] Bridgeman, S., Goodrich, M.T., Kobourov, S.G., and Tamassia, R. 2000. PILOT: An Interactive Tool for Learning and Grading. In *Proceedings of the SIGCSE Technical Symposium on Computer Science Education*, Austin, TX, March 2000, ACM Press, 139-143.

[10] Brusilovsky, P. and Su, H.: Adaptive Visualization Component of a Distributed Web-Based Adaptive Educational System. 2002. In *Proceedings of the 6th International conference on Intelligent Tutoring Systems*, June 2002, LNCS 2363, Springer Verlag, 229-238.

[11] Cavalcante, R., Finley T., and Rodger, S.H. 2004. A Visual and Interactive Automata Theory Course with JFLAP 4.0, In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, VA, March 2004, 140-144.

[12] Dancik, G. and Kumar, A.N. 2003. A Tutor for Counter-Controlled Loop Concepts and Its Evaluation. In *Proceedings of Frontiers in Education Conference*, Boulder, CO, November 2003, Session T3C.

[13] Etheredge, J. 2004. CMeRun: Program Logic Debugging Courseware for CS 1/2 Students. In *Proceedings of 35th SIGCSE Technical Symposium on Computer Science Education*, March 2004, 22-25.

[14] Fernandes, E. and Kumar, A. 2005. A Tutor on Subprogram Implementation. In *The Journal of Computing in Small Colleges*, 20(4), May 2005, 36-46.

[15] Fernandes, E. and Kumar, A. 2004. A Tutor on Scope for the Programming Languages Course, In *Proceedings of 35th SIGCSE Technical Symposium*, Norfolk, VA, March 2004, 90-95.

[16] Franke, R.H. and Kaul, J.D. 1978. The Hawthorne experiments: First statistical interpretation. *American Sociological Review*, 43, 623-643.

[17] Hristova, M., Misra, A., Rutter, M, and Mercuri, R. 2003. Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. In *Proceedings of 34th SIGCSE Technical Symposium on Computer Science Education*, February 2003, 153-156

[18] Johnson, W.L. 1986. *Intention-based diagnosis of novice programming errors*. Morgan Kaufman, CA.

[19] Kashy, E., Sherrill, B.M., Tsai, Y.., Thaler, D., Weinshank, D., Engelmann, M., and Morrissey, D.J. 1993. CAPA, An Integrated Computer Assisted Personalized Assignment System. In *American Journal of Physics*, 61(12), 1124-1130.

[20] Koffman, E.B. and Perry, J.M. 1976. A Model for Generative CAI and Concept Selection. *International Journal of Man Machine Studies*. Vol. 8, 397-410.

[21] Kostadinov, R. and Kumar, A.N. 2003. A Tutor for Learning Encapsulation in C++ Classes, In *Proceedings of ED-MEDIA 2003 World Conference on Educational Multimedia, Hypermedia and Telecommunications*, Honolulu, HI, June 2003, 1311-1314.

[22] Krishna, A., and Kumar A. 2001. A Problem Generator to Learn Expression Evaluation in CS I and Its Effectiveness. In *The Journal of Computing in Small Colleges*, 16(4), May 2001, 34-43.

[23] Kumar, A.N. 2005. Results from the Evaluation of the Effectiveness of an Online Tutor on Expression Evaluation, In *Proceedings of 36th SIGCSE Technical Symposium*, St. Louis, MO, February 2005, 216-220.

[24] Kumar, A.N. 2004. Generation of Demand Feedback in Intelligent Tutors for Programming. In *Proceedings of The Seventeenth Canadian Conference on Artificial Intelligence*, London, Ontario, Canada, May 2004, Lecture Notes in Artificial Intelligence 3060, Springer, 444-448.

[25] Kumar, A.N. 2002. A Tutor for Using Dynamic Memory in C++, In *Proceedings of 2002 Frontiers in Education Conference*, Boston, MA, November 2002, Session T4G.

[26] Kumar, A.N. 2002. Model-Based Reasoning for Domain Modeling in a Web-Based Intelligent Tutoring System to Help Students Learn to Debug C++ Programs, In *Proceedings of Intelligent Tutoring Systems*, Biarritz, France, June 2002, 792-801.

[27] Kumar A. 2001. Learning the Interaction between Pointers and Scope in C++, In *Proceedings of The Sixth Annual Conference on Innovation and Technology in Computer Science Education*, Canterbury, UK, June 2001, 45-48.

[28] Kumar A.N. 2000. Dynamically Generating Problems on Static Scope, In *Proceedings of the Fifth Annual Conference on Innovation and Technology in Computer Science Education*, Helsinki, Finland, July 2000, 9-12.

[29] Kumar, A.N., Schottenfeld, O. and Obringer, S.R. 2000. Problem Based Learning of Static Referencing Environment in Pascal, In *Proceedings of the Sixteenth Annual Eastern Small College Computing Conference*, Scranton, PA, October 2000, 97-102.

[30] Odekirk-Hash, E. and Zachary, J.L. 2001. Automated Feedback on Programs Means Students Need Less Help from Teachers. In *Proceedings of 32nd SIGCSE Technical Symposium on Computer Science Education*, February 2001, 55-59

[31] Reiser, B., Anderson, J. and Farrell, R. 1985. Dynamic student modelling in an intelligent tutor for LISP programming. In *Proceedings of International Joint Conference on Artificial Intelligence*. Los Altos, CA, August 1985, 8-14.

[32] Shah, H. and Kumar, A.N. 2002. A Tutoring System for Parameter Passing in Programming Languages. In *Proceedings of The Seventh Annual Conference on Innovation and Technology in Computer Science Education*, Aarhus, Denmark, June 2002, 170-174.

[33] Singhal N., and Kumar A. 2000. Facilitating Problem-Solving on Nested Selection Statements in C/C++. In *Proceedings of Frontiers in Education Conference,* Kansas City, MO, October 2000, IEEE Press, Session T4C.

[34] Weber, G. and Brusilovsky, P. 2001. ELM-ART: An Adaptive Versatile System for Web-Based Instruction. *International Journal of Artificial Intelligence in Education*, 12, 351-384.

```
program main;
        var q: Real;

        procedure chi;
                var z: Real;
                var y: Char;
                var p: Boolean;

                procedure lambda;
                        var s: Real;
                begin {lambda}
                        ...
                end   {lambda}
        begin {chi}
                ...
        end   {chi}

        procedure sigma;
                var q: Boolean;

                procedure psi;
                        var a: Char;
                        var b: Real;
                begin {psi}
                        ...
                end   {psi}
        begin {sigma}
                ...
        end   {sigma}

        procedure delta;
                var x: Real;
                var b: Boolean;

                procedure tau;
                        var q: Char;
                        var s: Boolean;
                        var a: Boolean;

                        procedure rho;
                        begin {rho}
                                ...
                        end   {rho}
                begin {tau}
                        ...
                end   {tau}

                procedure theta;
                        var z: Char;
                begin {theta}
                        ...
                end   {theta}
        begin {delta}
                ...
        end   {delta}
begin {main}
        ...
end   {main}
```

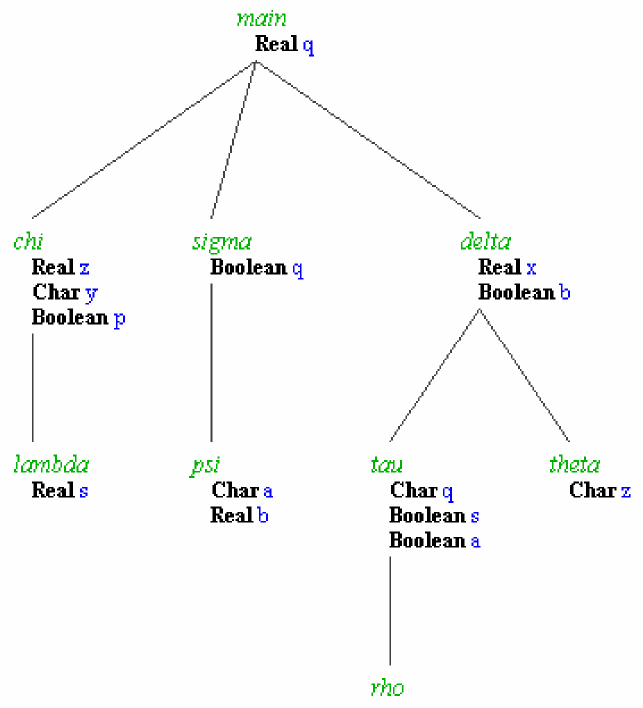
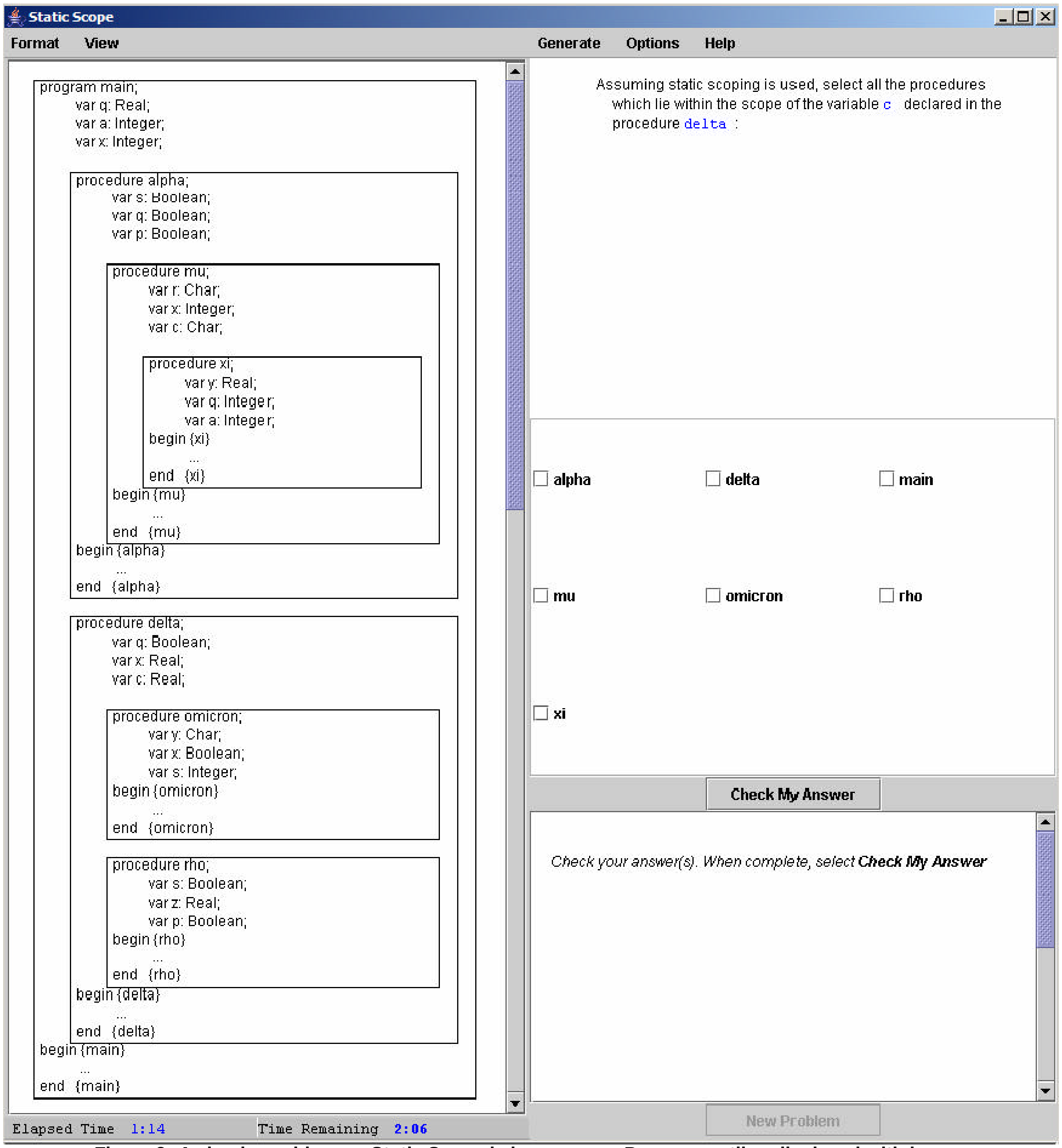


Figure 2: The outline of a program and the static tree to which it corresponds – both generated by the tutor



```
program main;
    var q: Real;
    var a: Integer;
    var x: Integer;

    procedure alpha;
        var s: Boolean;
        var q: Boolean;
        var p: Boolean;

        procedure mu;
            var r: Char;
            var x: Integer;
            var c: Char;

            procedure xi;
                var y: Real;
                var q: Integer;
                var a: Integer;
            begin {xi}
                ...
            end  {xi}
        begin {mu}
            ...
        end  {mu}
    begin {alpha}
        ...
    end  {alpha}

    procedure delta;
        var q: Boolean;
        var x: Real;
        var c: Real;

        procedure omicron;
            var y: Char;
            var x: Boolean;
            var s: Integer;
        begin {omicron}
            ...
        end  {omicron}

        procedure rho;
            var s: Boolean;
            var z: Real;
            var p: Boolean;
        begin {rho}
            ...
        end  {rho}
    begin {delta}
        ...
    end  {delta}
begin {main}
    ...
end  {main}
```

Assuming static scoping is used, select all the procedures which lie within the scope of the variable `c` declared in the procedure `delta` :

☐ alpha ☐ delta ☐ main

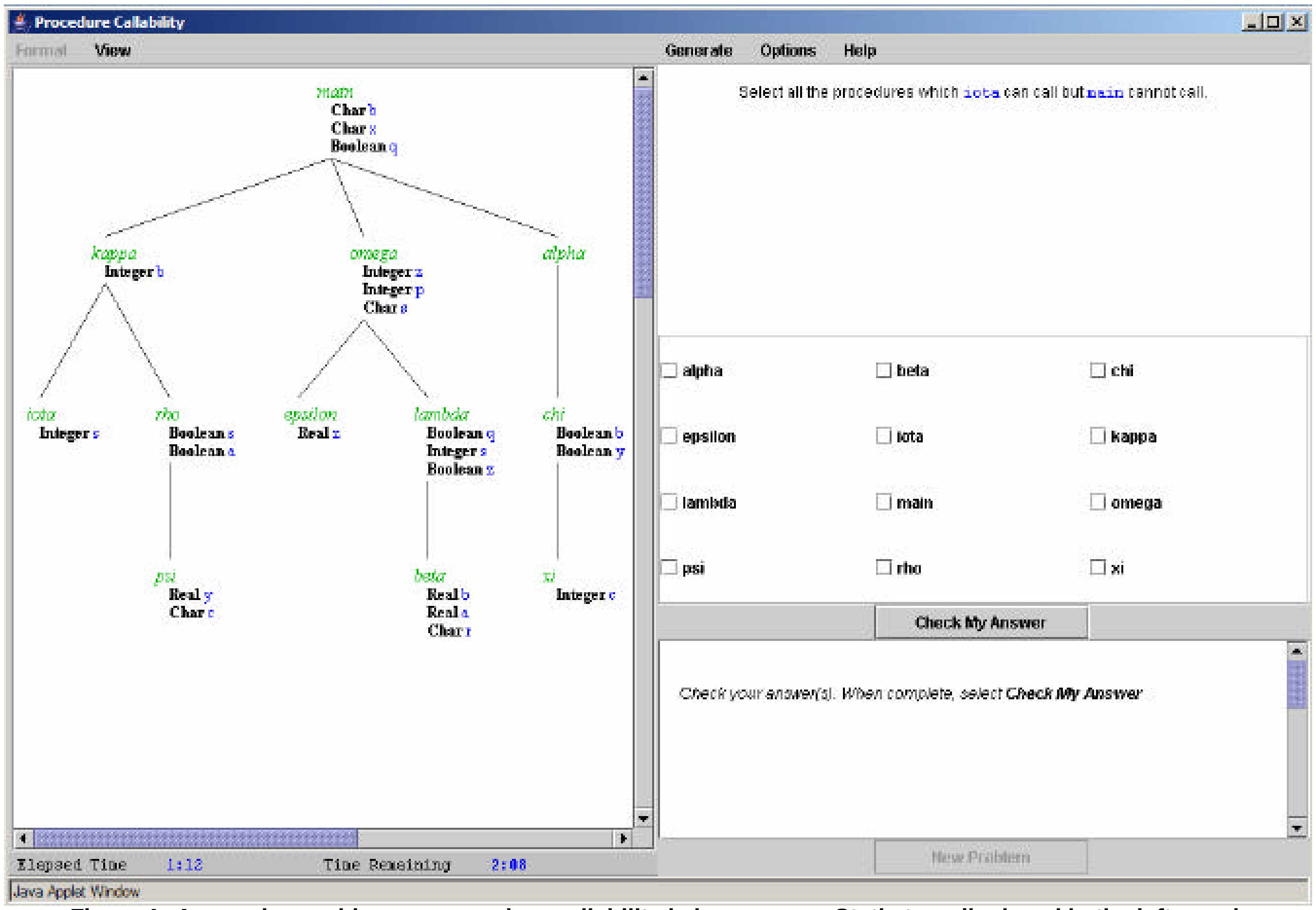
☐ mu ☐ omicron ☐ rho

☐ xi

Check your answer(s). When complete, select ***Check My Answer***

```
Elapsed Time   1:14          Time Remaining   2:06
```

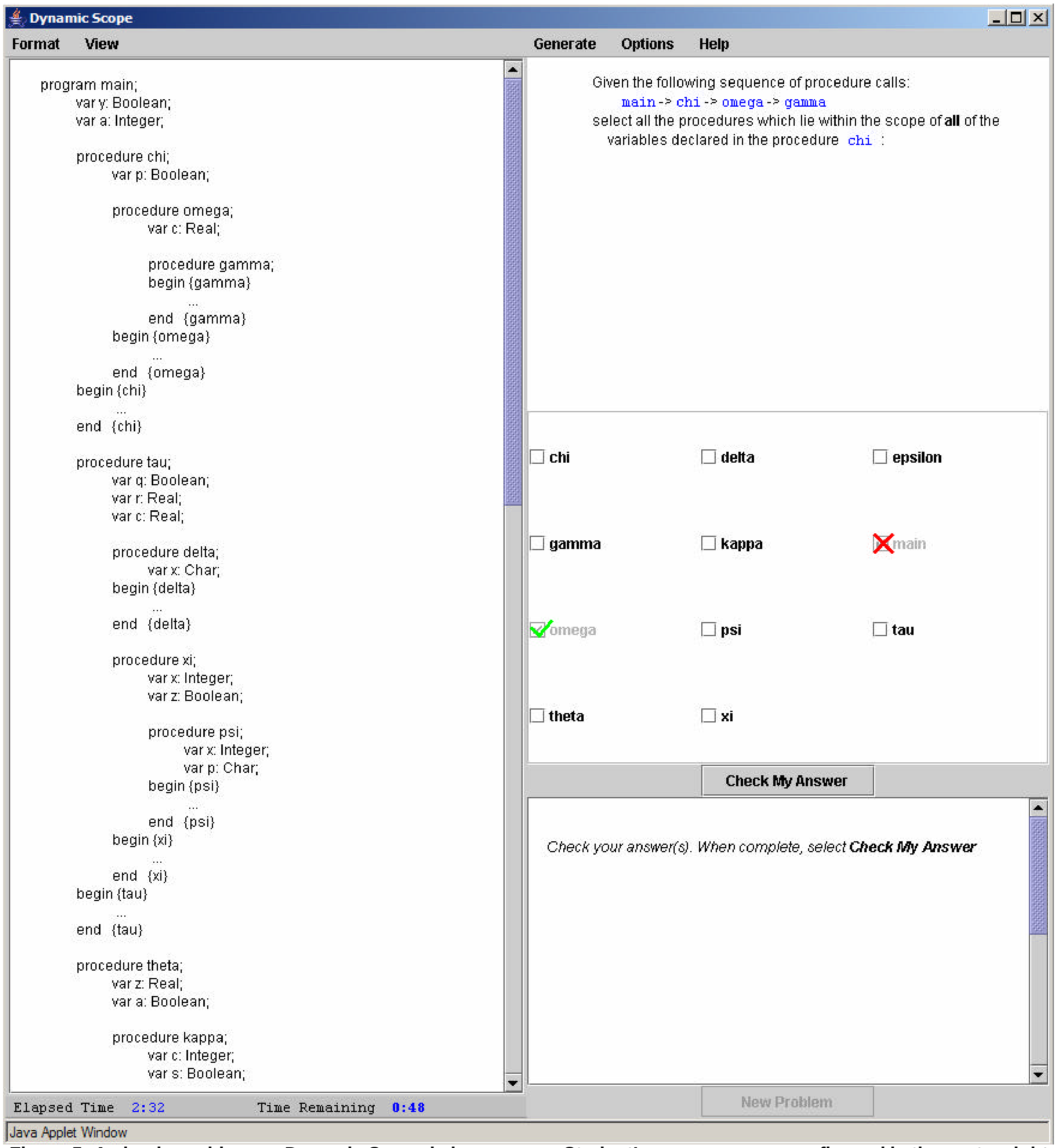
Figure 3: A simple problem on Static Scope is in progress. Program outline displayed with boxes.



Select all the procedures which iota can call but main cannot call.

*Check your answer(s). When complete, select **Check My Answer***

Figure 4 : A complex problem on procedure callability is in progress. Static tree displayed in the left panel.



```
program main;
    var y: Boolean;
    var a: Integer;

    procedure chi;
        var p: Boolean;

        procedure omega;
            var c: Real;

            procedure gamma;
            begin {gamma}
                ...
            end  {gamma}
        begin {omega}
            ...
        end  {omega}
    begin {chi}
        ...
    end  {chi}

    procedure tau;
        var q: Boolean;
        var r: Real;
        var c: Real;

        procedure delta;
            var x: Char;
        begin {delta}
            ...
        end  {delta}

        procedure xi;
            var x: Integer;
            var z: Boolean;

            procedure psi;
                var x: Integer;
                var p: Char;
            begin {psi}
                ...
            end  {psi}
        begin {xi}
            ...
        end  {xi}
    begin {tau}
        ...
    end  {tau}

    procedure theta;
        var z: Real;
        var a: Boolean;

        procedure kappa;
            var c: Integer;
            var s: Boolean;
```

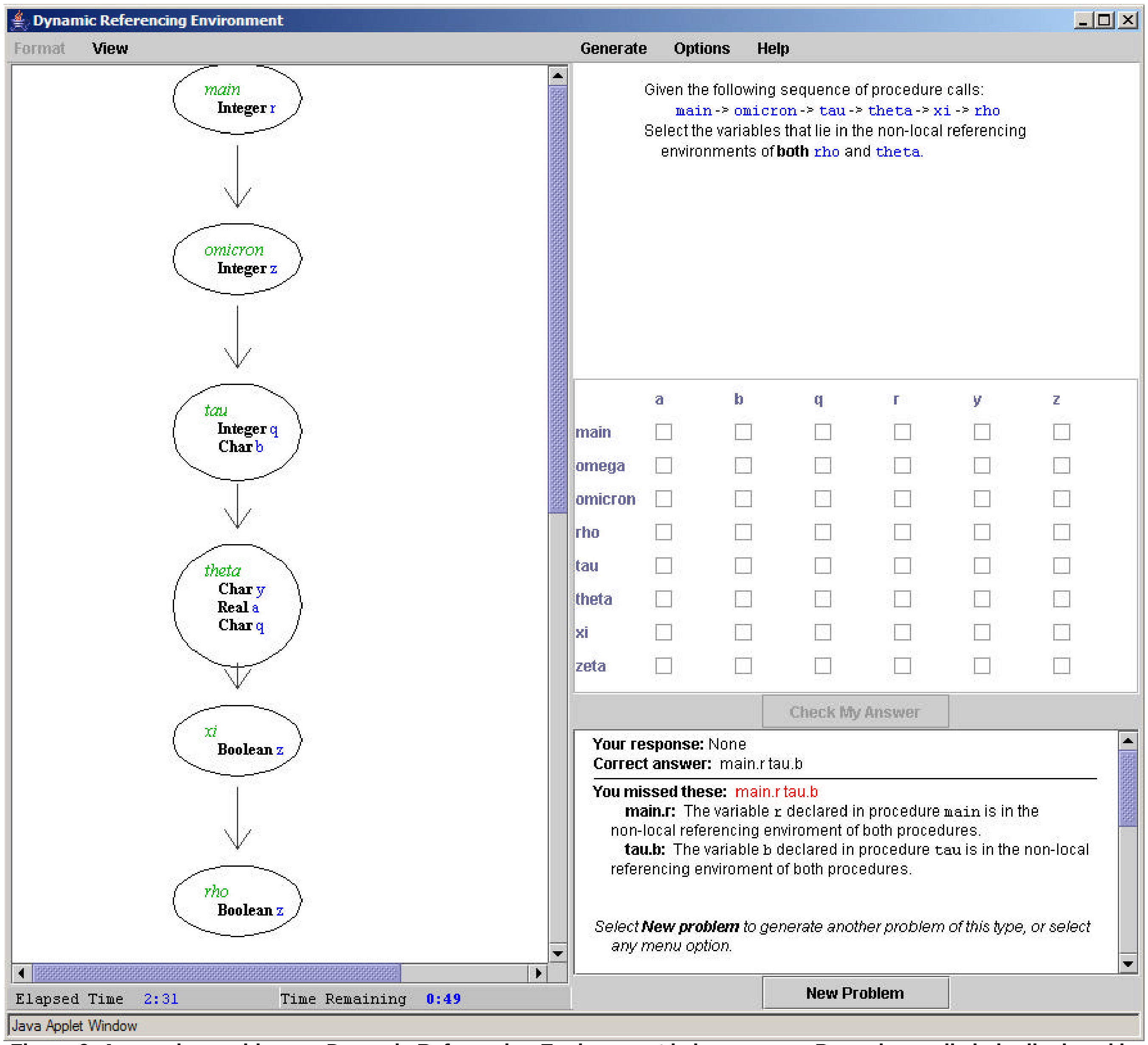
Given the following sequence of procedure calls:
main -> chi -> omega -> gamma
select all the procedures which lie within the scope of **all** of the variables declared in the procedure chi :

☐ chi	☐ delta	☐ epsilon
☐ gamma	☐ kappa	✗ main
✓ omega	☐ psi	☐ tau
☐ theta	☐ xi	

*Check your answer(s). When complete, select **Check My Answer***

```
Elapsed Time   2:32          Time Remaining   0:48
```

Figure 5: A simple problem on Dynamic Scope is in progress. Student's answers are error-flagged in the center right panel.



Given the following sequence of procedure calls:
`main -> omicron -> tau -> theta -> xi -> rho`
Select the variables that lie in the non-local referencing environments of **both** `rho` and `theta`.

	a	b	q	r	y	z
main	☐	☐	☐	☐	☐	☐
omega	☐	☐	☐	☐	☐	☐
omicron	☐	☐	☐	☐	☐	☐
rho	☐	☐	☐	☐	☐	☐
tau	☐	☐	☐	☐	☐	☐
theta	☐	☐	☐	☐	☐	☐
xi	☐	☐	☐	☐	☐	☐
zeta	☐	☐	☐	☐	☐	☐

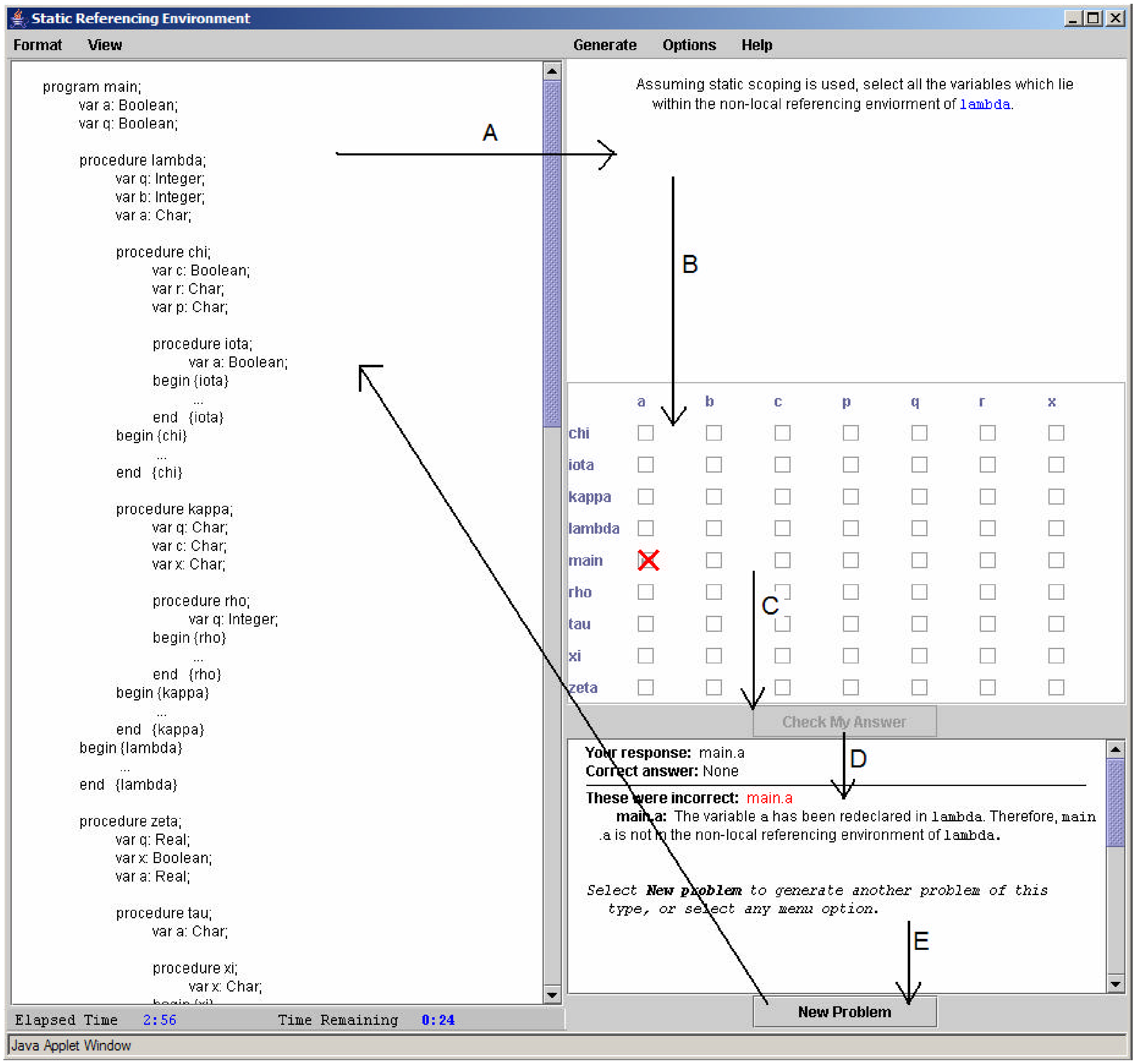
Your response: None
Correct answer: main.r tau.b

You missed these: main.r tau.b
main.r: The variable `r` declared in procedure `main` is in the non-local referencing enviroment of both procedures.
tau.b: The variable `b` declared in procedure `tau` is in the non-local referencing enviroment of both procedures.

*Select **New problem** to generate another problem of this type, or select any menu option.*

```
Elapsed Time   2:31          Time Remaining   0:49
```

Figure 6: A complex problem on Dynamic Referencing Environment is in progress. Procedure call chain displayed in the left panel. Demand feedback is displayed in the bottom right panel.



```
program main;
        var a: Boolean;
        var q: Boolean;

        procedure lambda;
                var q: Integer;
                var b: Integer;
                var a: Char;

                procedure chi;
                        var c: Boolean;
                        var r: Char;
                        var p: Char;

                        procedure iota;
                                var a: Boolean;
                        begin {iota}
                                ...
                        end   {iota}
                begin {chi}
                        ...
                end   {chi}

                procedure kappa;
                        var q: Char;
                        var c: Char;
                        var x: Char;

                        procedure rho;
                                var q: Integer;
                        begin {rho}
                                ...
                        end   {rho}
                begin {kappa}
                        ...
                end   {kappa}
        begin {lambda}
                ...
        end   {lambda}

        procedure zeta;
                var q: Real;
                var x: Boolean;
                var a: Real;

                procedure tau;
                        var a: Char;

                        procedure xi;
                                var x: Char;
```

Assuming static scoping is used, select all the variables which lie within the non-local referencing enviorment of `lambda`.

	a	b	c	p	q	r	x
chi	☐	☐	☐	☐	☐	☐	☐
iota	☐	☐	☐	☐	☐	☐	☐
kappa	☐	☐	☐	☐	☐	☐	☐
lambda	☐	☐	☐	☐	☐	☐	☐
main	☒	☐	☐	☐	☐	☐	☐
rho	☐	☐	☐	☐	☐	☐	☐
tau	☐	☐	☐	☐	☐	☐	☐
xi	☐	☐	☐	☐	☐	☐	☐
zeta	☐	☐	☐	☐	☐	☐	☐

Your response: main.a
Correct answer: None

These were incorrect: main.a
main.a: The variable `a` has been redeclared in `lambda`. Therefore, `main.a` is not in the non-local referencing environment of `lambda`.

*Select **New problem** to generate another problem of this type, or select any menu option.*

Elapsed Time 2:56 Time Remaining 0:24

Figure 8: The flow of action in the tutor. A problem on Static Referencing Environment is in progress.