Dynamically Generating Problems on Static Scope

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

Abstract
Solving problems is an integral part of learning in Computer Science. In order to provide students with a vast supply of problems with which to practice, we propose to use applets that automatically generate problems. In this paper, we first discuss the capabilities required of such applets, and then, present the design and features of an applet we have developed to automatically generate problems on static scope in Pascal.

1 Introduction
Many topics in Computer Science are best learned by solving problems. E.g., scheduling algorithms in Operating Systems, inside-outside tests in Computer Graphics, and scope/referencing environment in Programming Languages. Since textbooks usually include only a limited supply of problems, we have been interested in developing an automatic source of problems, a program that dynamically generates problems on a given topic. When implemented as Java applets, such programs can be made widely available and accessible around the clock.

Applets to generate problems may be used in several ways in a course:
1. Problem-based learning improves long-term retention [3], and is better than traditional instruction for improving the ability of students to solve real-life problems. But, it tends to reduce initial levels of learning, because it is time-consuming both inside and outside the classroom [10]. Applets may be used in a course to reduce the time spent inside the classroom on problem-based learning. Instructors may require their students to solve a certain number of problems after class as part of the course. Students may use such applets to practice solving problems, and improve their understanding of the concepts.
2. Instructors may use such applets to design problems for assignments and tests. This is especially useful to the instructors who teach a course year after year, and are reluctant to reuse problems from one semester to the next for various reasons. They may use such applets to automatically generate problems, validate their correctness, and ensure that they are set at the right level of hardness, all of which are time-consuming tasks.
3. Instructors may also use the applets to administer online tests that involve solving problems. By randomizing the problems generated by an applet, and serving a different problem instance to each student, instructors can thwart cheating during a test.
4. Finally, the applets may be used as an integral component of distance learning courses, wherein, active online learning is critical to the success of a program.

We refer to applets designed to generate problems on a given topic as problets.

2 Capabilities of a Proplet
We have identified the following as the capabilities a typical proplet must possess, irrespective of its content:

- A proplet must be capable of automatically generating problems. Systems that generate problems have been discussed in Intelligent Tutoring Systems community for at least two decades (e.g., [1, 5, 11]). In such systems, problems may be either generated dynamically, or merely selected from a compiled repertoire. We believe that, in order to minimize the chances of a proplet generating the same problem more than once for a student, the proplet must generate the problems dynamically. Such non-repetitive nature is needed if we want students to be able to use the same proplet to practice as well as take tests. In addition,
  - a proplet must be able to solve the problems it generates;
  - a proplet must be able to explain where and why the user's response is incorrect.
  - a proplet must be able to change the level of difficulty of the generated problems to suit the needs of the user.
A problet must include the visualization techniques normally available to the user if the user were to solve the problems with pen and paper. In other words, the on-line problem-solving environment must not be inferior to the traditional problem-solving environment.

The technology behind the problet must be transparent to the user, so that the user can concentrate on solving problems and not be hindered by having to do so on-line.

Finally, the problet may provide tools not typically available off-line, and thereby, enhance the problem-solving environment. E.g., searching and sorting facilities.

In addition, the problet should ideally be available across platforms and without the need for proprietary tools. Java applets readily meet this requirement.

3 The Design of Problets

We have identified the following steps in designing a problet:

1. A template is designed for the problem, bearing in mind the concepts that must be tested in the problem, and the information that must be provided as part of the problem. The template itself includes the following components:
   - The background information necessary to solve the problem.
   - The stem of the problem, which states the problem. It must be self-contained.
   - The response options provided to the user.
   - The format of the feedback to be provided to the user: feedback to accompany correct responses, and feedback to accompany incorrect responses.

2. Problems are randomly generated based on the template. In order to do so, we first identify invariant and variable parameters in the template and set their values appropriately.

   Invariant parameters of a template are the variables that determine how problems are generated. They may be used to control the hardness, type, etc. of the generated problems. Variable parameters of a template determine the current instance of the problem being generated. Their values are randomized, so that a different problem is generated each time.

   Heuristics may have to be used not only to prevent the generation of trivial problems, but also to promote the generation of "harder" problems. We use the following heuristic definition of hardness of problems for our purposes: "The more the plausible response options a user has for a problem, the harder the problem" [6,7].

3. Visualization tools must be designed and built into the problet. These tools may be included not only to provide the facilities to which the user has access when solving problems off-line, but also to provide facilities easily available on-line, that augment the problem-solving environment.

   A well-designed problet may provide several alternative visualizations of a problem (text, graph, etc.), formatting options for these visualizations that help the user focus on the highlights of the problem, and facilities for the user to interact with the visualizations.

   Human-factors issues must be taken into account when designing the visualizations. E.g., a problet may display no more than 9 items on the screen at any time, as recommended by Miller's rule of seven plus/minus two [8]. (This rule may also be used during problem generation to limit the cognitive load on the user.)

In the rest of the paper, we will describe the problet we have designed to dynamically generate problems on the topic of static scope.

4 Problet on Scope

Scope is one of the properties of variables in programming languages. The issue of scope (static or dynamic) arises in all the programming languages. In the Programming Languages course, static scope is typically studied in the context of Pascal programs [9]. Pascal is considered the ideal language to study static scope because it allows nesting of procedure definitions, and therefore, it is significantly more complicated than most other procedural languages. We will discuss the implementation of a problet for static scope in Pascal.

The scope of a variable in a Pascal program is said to be the most immediately enclosing block, minus any enclosed block where the variable has been re-declared. In order to identify the scope of a variable in a Pascal program, one has to first establish the nesting structure of its procedures. This is a tree structure, with the main program as its root, and nested procedures as nodes in the tree. It is called the static tree of the program. Once a student draws this tree for a Pascal program, identifying the scope of a variable declared in a procedure is as easy as traversing the sub-tree of the procedure until either the leaf nodes are reached, or procedures where the variable has been re-declared.

In the rest of this section, we will describe several aspects of our problet on static scope: the template for the generated problems, randomization of problem generation and tools for visualization. Note that for the sake of convenience, we will often describe aspects of the problet in terms of the static tree of a program, rather than the program itself.

4.1 Problem Generation Template

Background Information: A Pascal program with nested procedure definitions is randomly generated, and displayed
in a separate window. Within each procedure, only variable declarations and nested procedures are listed. The syntactic structure of each procedure is:

```
procedure name
  <variable declarations>
  <nested procedure definitions>
begin {name}
  ...
end {name}
```

Based on this program, the problet asks the user to identify the scope of a randomly chosen variable declared in a randomly chosen procedure in the program.

**Problem Stem:** The stem of the problem, along with the response options is displayed in a second window. The stem reads: "Assuming static scoping is used, select all the procedures which lie within the scope of variable a declared in the procedure main." Note that only the variable name and procedure name are changed from one instance of the problem to the next.

**Response options:** The problet lists the names of all the procedures in the program as checkboxes (See Figure 1). The user is expected to check all the procedures which (s)he believes are within the scope of the variable, and click on a button called "Check My Answer," when done.

**4.2 Randomization of Problem Generation**

**Background Information:** All the following aspects of the Pascal program are determined randomly: the number of procedures at each level in the static tree, the names of the procedures, the number of variables declared in each procedure, and the names and data types of these variables.

The number of procedures at a given level in the static tree is determined as follows: main, which is the root of the static tree may have either 2 or 3 children. Each of its children may have 1 or 2 children, each of its grandchildren may have none or one child, and its great-grandchildren may not have any children. Therefore, the maximum number of children a procedure may have is inversely proportional to the level of the procedure in the static tree. This heuristic is used to promote the generation of trees that are broad at the root.

Each procedure may have a random number of variables in the range 0 to 6 - level, where level is the level of the procedure in the tree (main being at level 0). This heuristic is designed to promote the declaration of more variables per procedure at lower levels in the tree.

Procedure names and variable names are randomly chosen from a repertoire of names: procedure names are never repeated, but variable names are often re-used. The data types of the variables are randomly chosen from among the four primitive data types in Pascal: Integer, Real, Boolean and Char.

**Problem Stem:** The variable whose scope is sought in the problem, and the procedure in which it is declared are randomly selected from the program. The procedure is more likely to be selected from a lower level of the static tree (e.g., main) than from say, leaf nodes. This heuristic, coupled with the earlier heuristics for generating broad trees with more variables at lower levels of the tree, is designed to generate problems that have many correct responses. Therefore, according to the definition we use for hardness of problems, these heuristics tend to generate harder problems. Our strategy is similar to using context-free probabilistic grammar for problem generation [4].

**Response options:** For implementation reasons, currently, we list checkboxes for procedures in the depth-first order of the procedures in the static tree. However, this order of listing procedures in a program is inadvisable for static scope, because the children of a procedure (and hence, likely answers to a problem) are listed consecutively after the procedure. This may give away the answers to the user, and enable the user to correctly solve problems without quite understanding the concept of static scope. In the future, we plan to list the procedures in alphabetical order.

**4.3 Solving Problems & Feedback**

The problet is capable of solving the generated problems. It does so by traversing the static tree in a depth-first fashion. When the user clicks on "Check My Answer" button in the template, the problet solves the problem and compares its solution with the user's response. It then provides feedback in the following form:

- It first lists the procedures selected by the user, the procedures in the correct answer, and the procedures selected by the user which are correct;
- Next, it lists the procedures that the user missed. Note that:
  
  Missed procedures = Procedures in correct answer - Procedures in user's response

For each missed procedure, it lists the ancestors of the procedure and points out that the variable in question was not re-declared in any of these procedures. E.g., in a problem where the scope of variable 'a' declared in main is in question, the problet prints the following message for a missed procedure epsilon: "epsilon is a descendant of main, and a is not re-declared in any of the procedures: alpha, epsilon", assuming the procedure alpha is the parent of epsilon and the child of main (See Figure 1).

- Next, it lists the procedures selected by the user that are incorrect. Note that
  
  Incorrect procedures = Procedures in user's response - Procedures in correct answer

For each incorrect procedure, it lists one of the following explanations as to why the procedure is not in the scope:
The procedure is not a descendant of the procedure in question (e.g., "delta is not a descendant of alpha")

The variable in question was re-declared in the incorrectly selected procedure or one of its ancestors (e.g., "a has been re-declared in omega. Therefore, zeta is not in the scope of a declared in main", assuming procedure omega is the parent of zeta).

4.4 Visualization

The problet provides the following visualization tools to clarify the presented code:

- the user may print the code using a different color for each level of nesting;
- the user may print the code with a bounding box around each procedure (See Figure 1).

In addition, the user may change the level of indentation in the code.

5 Implementation

We have implemented this problet in Java 2 with Swing. We have used the Model-View-Controller pattern [2] to separate the model of the problem from the interface and interactivity of the problet. We hope that this separation will make it easier for us to revise and reuse the code. The problet is now posted on the Web at: http://orion.ramapo.edu/java/opl/scope/ and is being used in our Programming Languages course. We plan to evaluate the effectiveness of using problets to teach and learn problem-solving in this course. We also plan to build problets for related topics such as dynamic scope, referencing environment, scope implementation techniques, etc.

References


