ABSTRACT

Problem Solving is as important for learning Computer Science as programming. Yet, the choice of problems offered in textbooks, the traditional source of problems, is limited. In order to provide students with a vast supply of problems, we propose to build Java applets that automatically generate problems and provide necessary feedback to the user. We refer to such problem applets as problets. In this paper we will discuss the design and features of a problet for static referencing environment in Pascal.

1. INTRODUCTION

Most topics in Computer Science are best learned by solving problems, e.g., search algorithms in Artificial Intelligence, transformations in computer graphics and constructs of programming languages. The more the numbers and types of problems solved by a student, the better his/her understanding of the concept and its adaptability to real-life situations.

To solve more and varied types of problems, students must have access to a large set of such problems. The source of the problems in most cases is generally the textbook, but books have their limitations. The number of problems is limited by the space available in the book. These problems are often recycled from one edition to the next and often, solution manuals for the problems can be obtained by students either online or from publishers. The textbooks are not capable of providing detailed feedback on whether and why a student's answers are correct/wrong. Textbooks do not provide animations and visualizations that may be helpful in solving a problem. Therefore, support for problem-solving in traditional textbooks is limited.

In order to address these concerns, we propose to build Java applets that automatically generate problems. We refer to such applets as problets. In the next section, we will briefly discuss problets in general. In Section 3, we will discuss the design and implementation of a problet for static referencing environment in Pascal.
2. PROBLETS

A typical problet must be capable of generating problems automatically and should preferably not generate the same problem more than once during a session. The problet must be capable of solving the problems that it generates, providing feedback to the user and calibrating the hardness of the problem based on user’s needs.

A problet must provide visualization techniques that assist the user in solving the generated problems. These techniques must provide transparency of technology and preferably provide tools that will enhance the problem-solving environment. They must also take into account human factors issues (e.g., seven plus two rule [3]).

A typical problet employs a problem template, which is the blueprint of the types of problems that will be generated by the problet. The template consists of the following components:

- The background information necessary to solve the problem;
- The stem of the problem, which states the problem;
- The response options provided to the user;
- The format of the feedback provided to the user.

A problet generates problems based on the template, by randomizing the parameters of the template. Heuristics may be used during problem generation to control the hardness of the generated problems.

We use the following definition of hardness when generating problems: "The more the plausible response options a user has for a problem, the harder the problem" [1,2]. E.g., multiple choice problems are harder than true/false problems, and problems with free-form answers are harder than multiple choice problems. Our interest is in generating multiple choice problems.

3. PROBLET ON STATIC REFERENCING ENVIRONMENT

The non-local referencing environment of a procedure may be determined dynamically at run time or statically at compile time. Pascal is one of many languages that determine their referencing environment statically. Pascal is often used to study static scope because it allows procedure definitions to be nested [4].

The non-local referencing environment of a procedure in Pascal consists of all the variables from its ancestors that have not been re-declared locally. Students have trouble identifying the non-local referencing environment of a Pascal procedure. The problet we describe in this section will assist the user in understanding non-local static referencing environment in Pascal.

In order to identify the non-local referencing environment of a procedure in Pascal, one has to first establish the nesting structure of its procedure definitions. This is a tree structure with main as its root, and nested procedures as nodes in the tree. It is called
the static tree of the program. Identifying the non-local referencing environment of a procedure simply involves traversing the static tree from the procedure up to the root. Any variable that is encountered for the first time during this traversal is included in the non-local referencing environment of the procedure. Note that for the sake of convenience, we will often describe aspects of the prole in terms of the static tree of a program rather than the program itself.

We will now describe the design of a prole to generate problems on non-local referencing environment in Pascal. We will describe the background information, the stem, the response options, feedback and visualization features of the prole.

**Background Information:** The prole first generates the outline of a Pascal program and displays it in a separate window. The syntactic structure of a Pascal procedure is of the form:

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

A typical Pascal program with several levels of nesting may read as follows:

```
program main;
    <variable declarations>
    procedure level1-procedure;
        <variable declarations>
        procedure level2-procedure;
            <variable declarations>
            begin {level2-procedure}
                ...
            end {level2-procedure};
        begin {level1-procedure}
            ...
        end {level1-procedure};
    begin {main}
        ...
    end {main}.
```

All the following aspects of the Pascal program are determined randomly by the prole: the number of procedures at each level, the names of the procedures, the number of variables declared in each procedure and the names and data types of these variables.

The root of the static tree is main. The names of the other procedures are chosen randomly, and are not repeated. The names and types of the variables in a procedure are chosen randomly. A variable may not be declared more than once in a procedure, but it can appear in several procedures at the same time. We use heuristics to generate more variables per procedure at lower levels of the static tree.
**Problem Stem:** The stem of the problem, along with response options is displayed in a second window. The stem reads “*Assuming static scoping is used, select all the variables which lie within the non-local referencing environment of procedureName (Check all the correct boxes then click on Done.)*”

The procedure in the stem is chosen randomly. We use heuristics to select procedures that are deeper in the static tree. Combined with the earlier heuristic, this leads to generation of problems that are harder to solve according to our earlier definition, i.e., the selected procedures will have a large number of variables in their non-local referencing environment.

**Response Options:** The response options are a cross product of the list of variable names and the list of procedure names. A grid is used to display possible responses. The procedure names are used as rows and the variable names as columns. Both are alphabetized and check boxes are placed at the intersections to represent selections. A user could therefore select a variable that is not declared in a given procedure; the user could also select a variable multiple times, i.e., from multiple procedures. By not preventing these choices, the proplet avoids influencing the user’s response. We arrived at this design for the interface after considering others such as drop-down boxes and non-grid versions of checkboxes, which we thought, were not as easy or intuitive to use.

Sometimes a procedure may not have any non-local variables in its referencing environment. Although we could have provided a separate checkbox for the user titled "No non-local variables in the environment", we chose not to. Currently the user will simply not select any variables before submitting the response. This was again done to avoid influencing the user when he/she is solving the problem.

**Feedback:** When the user clicks on the “Check My Answer” button, the proplet verifies the user's response and displays the correct answer. It lists all the variables that the user correctly identified. For the variables the user missed entirely, or identified incorrectly, it lists why those variables should (should not) be in the non-local referencing environment of the procedure.

The user may choose to get another question on the same program by clicking on the “Get Another Problem” button. The proplet generates another question by randomly selecting the name of a different procedure in the Pascal program.

**Visualization:** The proplet 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.

These facilities help the user clearly see the nesting structure of procedures and analyze the non-local referencing environment of a procedure (See Figure 1).
- The user may also choose to view the static tree of the program. This graphical representation makes it much easier to analyze the code (See Figure 2).
4. FUTURE WORK

We have been implementing another prolet for static scope of variables[5]. We hope to integrate it with the current prolet and make both available to students in the Programming Languages Course.

We also plan to incorporate logging facilities into the prolet. This will help us assess the effectiveness of using the prolet in our course.

Figure 1. Screen shot of the Prolet: Generated code, Problem, Interface
Figure 2: Screen shot of the Problem: Visualization and Feedback

REFERENCES


