Searching for Examples with a Programming Techniques Editor

Paul Brna
Computer Based Learning Unit, Leeds University, Leeds, England

Searching through a library of examples for a similar task or similar solution is one way in which novice programmers learn to program. Providing help for novices to become more proficient programmers entails helping them both to see the significance of the 'deep' features of the current task and to take advantage of them both in searching a library of examples, and in selecting and using an appropriate case.

In this paper the focus is primarily on the problem of accessing a suitable case. For detailed consideration, a programming environment is utilised that features an 'intermediate description language'. This environment is SunTed, a Prolog Techniques Editor — which provides novice support of various types, including facilities to retrieve cases.

The basic issues addressed are: whether or not an intermediate description language for cases is a suitable means of supporting novices in their learning to program; the nature of the fundamental constituents of an intermediate description language; what can be learned from a system that implements a specific example of such an approach; and the consequences for the design of systems that support learning.

Keywords: case based search and retrieval, intermediate description languages, Prolog techniques, program editors

1. Introduction

Searching through a library of examples for a similar task or similar solution is one way in which novice programmers learn to program. However, it is now well accepted that novices are inclined to index more readily on surface features of the task or solution than on 'deeper' features. For example, if a student is to flatten a list, then he/she might use the program for ordering a list of integers as the basis of a solution. Using the term "list" as an index is relatively superficial compared with using "tree" as the index since flattening a list entails managing a recursive data structure. However, little or no research has been undertaken to help novices search effectively for good examples.

The main problem for those interested in helping the novice to become a more proficient programmer is how to support and encourage the novice both to see the significance of these deeper features and to take advantage of them in searching a library of examples, and then in selecting and using an appropriate case.

The approach taken here to the problem of helping the novice is through an analysis of the different *learning opportunities* that can be provided. In the case under consideration, the following significant learning opportunities are identified:

1. Learning to look through a library of cases and retrieve an appropriate case (using 'shallow' or 'deep' criteria)
2. Learning to index tasks and solutions (using 'shallow' or 'deep' criteria)
3. Learning to apply a retrieved case (effectively or not)
4. Learning to discriminate between best and less good cases

These opportunities are the ones which parallel the basic stages involved in the life cycle of working with cases, namely: generating, storing, retrieving, and selecting. Each of these opportunities has been the focus of research which can inform the provision of automated support for novices — e.g. [Schult & Reimann, 1995, Reimann *et al*, 1993, Weber, 1995, Ross, 1987].

In this paper, the design issues involved in supporting novices are explored with respect to only one of the above learning opportunities: accessing a suitable case (item 1).

For the access problem, three kinds of cases are distinguished: cases actually generated by the novice (the use of such cases is sometimes referred to as 'remindings'), examples worked through on paper by the novice, and cases that have not been studied in any detail (but might have been seen) by the novice. These latter are the ones that might be encountered in an explanatory text. The stress here is on the cases that have not been studied by the novice.

For detailed consideration, a programming environment that features an "abstract description language"is examined. This environment is SunTed, a Prolog Techniques Editor — designed to support novices by providing facilities to retrieve cases. This is used to illustrate the possibilities.

The basic questions are:

- What is the role of an 'intermediate description language'?
- What are the fundamental constituents of an intermediate description language?
- Is an intermediate description language for cases a suitable means of supporting novices in their learning to program?
- What can be learned from a system that implements a specific example of such an approach (SunTed, one of the Ted family of Prolog Programming Techniques editors)?
- What are the consequences for the design of learning systems?

It is claimed here that current approaches are typically insufficient to provide the kind of support suggested by current cognitive theories of problem solving, and by current thinking on how to support learning. There is a need to provide desirable learning opportunities. Some indications about the kinds of facilities that are needed to support novice programmers in the development of their programming expertise are briefly provided. This approach is compared with that taken by Linn and her colleagues in the "Perspective Library" [Linn *et al*, 1992].

2. An Intermediate Representation for 'Deep' Program Structure

The problem of what constitutes a suitable description language is now addressed, illustrating this with reference to SunTed. Our aim in selecting any specific description language is that it should act as an intermediate representation for novices in their attempt to index cases in a deeper way. In this way novices can be helped in their transition to becoming intermediate or expert programmers. A priori, the description language: should not involve a big detour in learning; must be fairly easy to learn; and must be possible to build upon in future development. (Linn *et al*, for example, have provided anecdotal evidence for novices extending the sophistication of a similar simple intermediate description language [Linn *et al*, 1992].)

The indices used to retrieve cases ideally should be easy to develop. A sequence of three classes of concept that can be incorporated in an index scheme is provided for in SunTed. An outline of a natural sequence of intermediate description languages is now given, starting with the simplest scheme (and the simplest one to implement):

1. Various constraints on the inputs and outputs of predicates. These constraints are very familiar to intermediate Prolog programmers. They are constraints that make no direct reference to the code structure.
2. Constraints on the structure of the code: this incorporates the notion of Prolog Programming Techniques [Brna *et al*, 1991], a relative of Programming Plans [Soloway, 1986]. Detailed information can be derived from our study of Prolog Programming Techniques [Brna *et al*, 1991, Bowles & Brna, 1993]. For the purposes of this paper, it is sufficient to know that Prolog Programming techniques can be seen as ways of constructing (Prolog) programs in a language dependent manner. The notion generalises to other languages, hence this work is not specific to Prolog.
3. Constraints on the relationship of the code to the task being undertaken.

To take item 1 first, novice Prolog programmer's likely familiarity with a fairly primitive notion of dataflow is utilised. The assumption is that they can quickly learn to specify a predicate's arguments in terms of whether they are input or output (or otherwise), and a few basic datatypes (e.g. atoms, integers, lists of atoms etc.). If programmers can quickly learn to use such a description language then there is some hope that the overheads involved in learning to use the language will be worthwhile. SunTed (see below) utilises a scheme for describing such constraints.

Alternative ways of describing constraints can be used. A scheme that describes various relationships that must hold between two predicate arguments is outlined below. These constraints capture a fairly language independent notion of the dependencies that must exist between arguments — e.g. that each element of an input list must be a member of the output list for a program that computes the union of two lists. This style of description is *algorithmic* in nature. The techniques editor described below provides this kind of constraint as part of its intermediate description language. (Prolog Programming Techniques are represented and accessible through a form of 'cut and paste'.)

The issue of constraints on the structure of code can be described in various ways. Prolog Programming Techniques provide an effective constraint on the code — in terms of dataflow or in terms of control flow. Such techniques are language dependent and task independent [Brna *et al*, 1991], but an intermediate description language based on techniques is problematic, partly because, like Programming Plans [Soloway, 1986] and Design Patterns [Gamma *et al*, 1994], many of them are likely to be unfamiliar to novice programmers. So the version of a Techniques Editor described here does not use techniques as part of the intermediate description language for the search for useful examples.

An ideal model of how to set up the indices for case-based retrieval would be expected to include indices both for the task structure (item 3) and the solution structure (item 2). This is because learning to program involves —at the least— learning to write/transform code at the program language level and map the task level to the program language level. This process of linking entities in the task level to the program level was described by Pennington as building a situation model [Pennington, 1987].

Indices for solution structure can be constructed with the help of work on Prolog Programming Techniques, programming plans or other ways of abstracting the constraints on the code. However, defining indices for task structure is harder — Weber, for example, has developed an interesting, though labour intensive approach, effectively based on constructing a task decomposition for each program in the library [Weber, 1995].

The retrieval would then be constrained partly by the known structure of the task and partly by the anticipated constraints on the task solution (program code). This model of retrieval

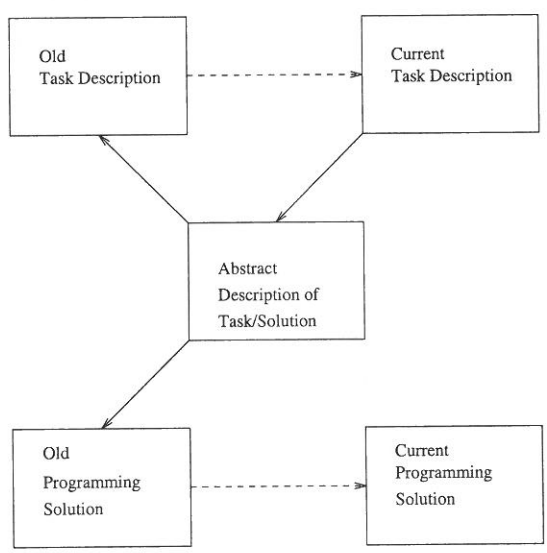

Fig. 1. Representing Case-Based Retrieval based on Task and Solution

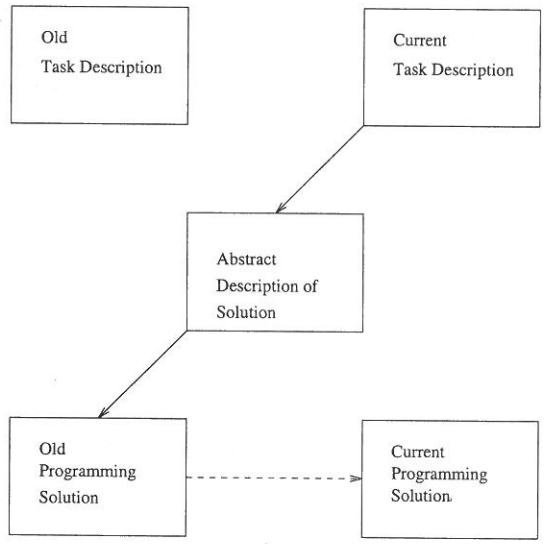

Fig. 2. Representing Case-Based Retrieval based on only the Solution

is represented crudely by figure 1 in which the dotted lines indicate some transfer of information while the solid lines indicate the path by which cases are retrieved.

A simpler model, the one used by the designers of SunTed, the Prolog Techniques Editor which is explored below, is based only on indices associated with the constraints on the anticipated solution — see figure 2. The consequence is that there is likely to be a greater cognitive load on the novice in terms of the work required to match the retrieved solution to generate a solution to the current problem. Such a situation will not directly encourage the student to see the current problem's deep structure — an aspect considered to be important in novice–expert transition [Chi *et al*, 1981].

A description is now given of a specific instance of an intermediate description language which was developed for use with an enhanced structure editor called SunTed [Bowles & Brna, 1993].

3. SunTed: A Case-Based Prolog Techniques Editor

SunTed is a Prolog Techniques Editor developed as part of a UK Joint Research Council grant entitled the "Construction and Evaluation of a Prolog Techniques Editor for Novices" and which originally involved researchers at the University of Edinburgh (Bowles, Brna, Pain and Robertson), the Open University (Kahney and Brayshaw) and Loughborough University (Ball, Ormerod and White). The original techniques editor, which we refer to here as MacTed, was developed for Macintosh computers.

Both SunTed and MacTed are based in part on the notion that novices should receive some assistance with the syntax of the language. Perhaps more importantly, SunTed and MacTed both provided support for users to exploit a class of Prolog Programming Techniques [Brna *et al*, 1991] connected with data flow [Bowles & Brna, 1993]. Although programming techniques are in some ways relatives of programming plans, it is argued in [Bowles & Brna, 1993] that programming plans possessed some undesirable features when compared with programming techniques.

As part of our work to explore the ways in which novices can be supported, the evaluation of the basic issues associated with the use of Prolog Programming Techniques has been undertaken. Brna analysed the problems likely to arise from trying to teach a techniques oriented approach in [Brna, 1993]. A techniques editor, MacTed, was developed which incorporated a subset of seven dataflow techniques that apply to single program clauses [Bowles & Brna, 1993]. This "structure editor" permitted a reasonable range of Prolog programs to be constructed. Empirical work was undertaken by Ormerod to examine the basis for the claim that techniques assisted novices in the implementation of programs [Ormerod & Ball, 1996]. The results of this study indicated that there appeared to

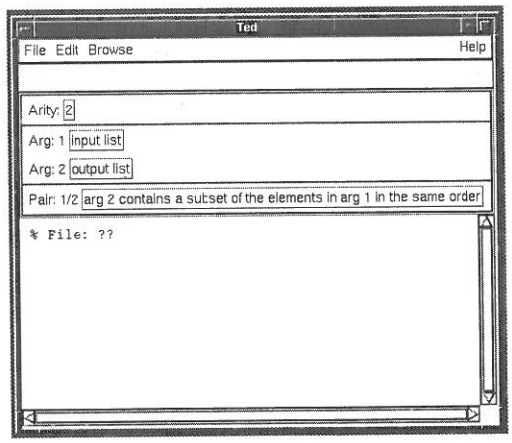

Fig. 3. SunTed's Editor with Browser Visible

be some advantages to the use of MacTed in conjunction with teaching Prolog. More specifically, evidence strongly supported the notion that teaching Prolog programming techniques with the help of MacTed promoted more effective coding of solutions to fairly straightforward examples.

SunTed was developed from MacTed using Tcl/Tk and SICStus Prolog v2.1.9, and redesigned for use with Unix-based workstations. The resulting editor has some unusual features which are the subject of this paper: an "intermediate description language" that could be used to retrieve cases from a case library, and a mechanism for appropriating relevant parts of the retrieved case. See figure 3 for SunTed's basic screen with the browser visible.

SunTed's basic screen has hotspots which permit the insertion of clauses, subgoals and arguments. In MacTed, the seven basic techniques are called up from a menu when entering details about the arguments — details of which are outside the scope of this paper. (SunTed has knowledge about the same set of techniques.) SunTed provides a significantly less restrictive interface, overcoming many of the problems encountered with MacTed and reported in [Ormerod & Ball, 1996]. However, the main innovation of the new version was the incorporation of the case-based retrieval mechanism.

Case-based retrieval is obtained by asking for the browser to be shown. Once activated, users are required to specify the arity (number of arguments) of a predicate. They are also required to provide the modes and types of each argument. For simplicity, the modes are described in terms of dataflow. The user can specify a mode as 'input' or 'output' (or both). The types allowed are restricted to integer, atom and list. All modes and types are selectable from a pull down menu. (Note that this is not quite the normal way in which the term 'mode' would be described in the Prolog community.)

Information can also be entered regarding the relationships that hold between the various arguments. These are also selectable via a pull-down menu. The permitted selections are restricted in a conservative manner to 'sensible' ones. For example, in the case of a predicate of arity 2 with one argument an input list and the other an output atom then the only permitted relationships concern whether or not the output argument is an element of the input list. The full range of relationships can be found in figure 5. An example of how the information may be entered is shown in figure 3.

Once the browser is requested to search for matches, the system uses case-based retrieval to find relevant programs. These might or might not be associated with a description of a task for which the program is a solution. Note that the search is over program descriptions and not over task descriptions. See figure 4 for an example of three retrieved cases with one (delete_one_1.pl) showing.

So far, nothing has been mentioned about how the case-based retrieval facilities interact with the techniques. In MacTed the techniques are explicitly invoked from mousing on a hot spot and using a pull down menu. This is not the case in SunTed, the version of Ted examined in

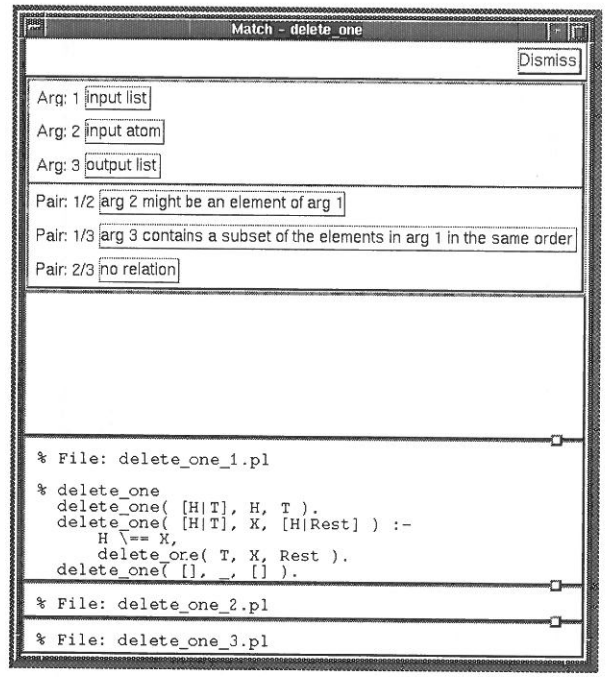

Fig. 4. Retrieval of a Relevant Case

this paper. There is no explicit mention of techniques in the interface. Rather, the notion of techniques can be exploited through the mechanism that the system provides for copying important features of the retrieved case.

The system provides a novel cut and paste mechanism. When certain parts of an example are double clicked the technique's structure is copied (this is not necessarily a 'continuous' piece of code). A double-click on any of the arguments of the retrieved case will copy a fragment of the case associated with one of the seven basic techniques. This 'copied' code can then be inserted as an argument into the program being constructed. In providing for copying fragments of the retrieved case into the program construction window SunTed does some limited automatic manipulations: the retrieved predicate name, for example, will be transformed to the name of the current predicate being constructed.

Techniques, therefore, only enter the picture when the programmer seeks to reuse components of the case retrieved – and, even then, they are not referenced by their names. In this way, the novice programmer may be able to gain some knowledge about the kinds of useful structure of which SunTed is aware without having to learn a great deal of terminology.

4. A Preliminary Study

In theory, the intermediate description language described might well help novice Prolog programmers — especially when embedded in the support environment provided by SunTed, but it was unclear as to whether or not the intermediate description language was usable.

The aim of the study was, therefore, to examine whether or not students with minimal Prolog experience (but a fair amount of programming experience) would be able to use the intermediate description language with any degree of precision.

4.1. Subjects

The subjects were 50 second year undergraduates taking a course on AI Programming at the Computing Department, Lancaster University during January and February of 1995. All subjects had received four lectures on Prolog but had no actual programming experience in Prolog, and no previous experience with SunTed. Their main programming language was ADA. None of them had seen the intermediate description language prior to the experiment.

4.2. Procedure

The subjects were given a document detailing the intermediate description language. They were asked to read the text, and then to fill in the desired information for ten questions (see appendix A). The time allowed was 40 minutes. At the end of this time, the anonymous responses were collected.

4.3. Method

The subjects were given the problem description

> **"Write a Prolog program *append* to append two lists together".**

as an example. They were informed that descriptions consisted of two parts. The first part consisted of a mode and type description for each of the arguments involved in a toplevel call.

> **In the example, *append* will need three arguments.**

The subjects were informed that a type is one of the tokens 'atom', 'number' or 'list', and a mode is either 'input' or 'output'. Thus each argument needed to be given one of the following descriptions:

input number	input atom	input list
output number	output atom	output list

They were informed that atoms were treated as including the set of numbers.

> **In the example, argument 1 is an 'input list', argument 2 is an 'input list' and argument 3 is an 'output list'.**

The subjects were informed that the second part consisted of a set of descriptions of relationships between pairs of argument in the toplevel call.

They were also told that relationships were statements of the form:

$$< argn >< relationship >< argm >$$

i.e., statements describing relationships between pairs of arguments. They were informed that it was not necessary to describe the relationship between every pair of arguments, nor was it necessary for all arguments to appear in the set of relationships; that is, the rôle of some arguments could be left undefined. The full range of descriptions as given to the subjects is found in figure 5.

The subjects were given the example for the *append/3* predicate.

> **In the example, *append* might be described by:**
>
> **argument 3 contains all the elements in argument 1 in the same order**
>
> **argument 3 contains all the elements in argument 2 in the same order.**

5. Results

For the 50 students who took part in attempting 10 questions, each within a fixed period of time, it was found that there were 430 items of possible 500 which featured full mode and type information for the procedure arguments. Of these 500 items, 411 featured an attempt to detail the expected relationships between arguments. Thus 82% of the items featured information about relations.

The implication is that students in this class found the description language connected with modes and types to be straightforward, and that most of them were able to define some plausible set of relationships that were expected to hold between the arguments for the final solution.

One approach to assessing the descriptions provided by the students was to submit these descriptions to SunTed itself. In its current form, SunTed has an initial database of 32 different tasks and 47 different programs (i.e. some tasks had more than one solution). Each of the tasks in the experiment had a program solution in the case library provided. A superficial measure of success was whether or not the student would

The Descriptions to Use
. . . is calculated from all the elements in . . .
. . . is calculated from a subset of the elements in . . .
(The term 'calculations' implies the computation of a number)
. . . is an index into . . .
. . . might be an element of . . .
. . . is an element of . . .
. . . is not an element of . . .
. . . is compared with . . .
elements of . . . are compared with elements of . . .
elements of . . . are compared with corresponding elements of . . .
. . . is a mapping of all the elements in . . .
. . . is a mapping of a subset of the elements in . . .
(The term 'mapping' indicates that each element of one argument is associated with a unique element of the other argument)
. . . is a sublist of . . .
. . . contains all the elements in . . . in the same order
. . . contains a subset of the elements in . . . in the same order
. . . contains all the elements in . . . in a different order
. . . contains a subset of the elements in . . . in a different order

The . . . indicates a slot that may or may not be filled. If it is filled, then it is intended that it should be with a reference to an argument e.g. **Argument 1**

Fig. 5. The descriptions permitted for relationships between arguments

have retrieved the correct solution (along with some others).

The ratio of finding to not finding the correct solution was examined — but only for legal entries (i.e. syntactically correct). The overall result was 271:28 or 9.68:1. Hence most students with legal entries were able to specify the solution well enough to retrieve the correct entry — possibly along with several others. This indicates a promising level of success — though not without some reservations to be discussed later.

The speculation was whether the ratio `find` to `not find` improved with regard to whether or not subjects who detailed relationships between arguments would do as well on this measure as those who did not detail any relationships at all. It was found that the `find:not find` ratio for those detailing relationships was 9.8:1 while for those not detailing any relationships this ratio was 9.4:1.

Though this result needs very careful interpretation, at first sight, in our experimental context, it made very little difference as to whether students used the simpler mode and type information for retrieval or the full description language. This issue was then studied using a different measure: students were selected who provided full information about modes and types and at least some information about relationships (the `modes+types+rels` condition). Their actual performances on 'recall' and 'precision' were compared with the performance that they would have achieved if they had not entered any information about argument relationships — i.e. just modes and types (the `modes+types` condition).

The measure of recall used for the data is the average value of the reciprocal of the number of results retrieved. Very accurate recall will therefore approach unity. The measure of precision used for the aggregated data is the number of searches which include the target code in the set of results retrieved. Maximum precision is therefore the number of elements in the dataset.

Of the 500 possible items, 314 satisfied the basic requirement — i.e. full information about modes and types. For the

straight comparison, the `find:not find` ratio for the `modes+types+rels` condition a figure of 105:209 (approx 1:2) is obtained. For the `modes+types` condition the ratio 289:25 (approx 11.6:1) was obtained. This argues strongly that the system will be difficult to use if the student is free to enter as many relationships as possible. As it happens though, SunTed only allows a single relationship between variables. Even so, entering up to 6 relationships for a relationship of arity 4 may prove a) time consuming and b) might be counter productive unless some other benefits of defining such relationships can be identified.

A further investigation was undertaken into the way in which the somewhat unusual measure of precision being used was affected by gradually increasing the number of relationships permitted. The ratio went from 105:209 for all relationship information present to 198:116 for dropping a single relationship at random to 264:50 if two relationships were dropped (if there was only one relationship specified then it would be dropped). From then on, the precision was generally good.

The algorithm was then modified to drop more informative relationships first. The results on the measure of 'precision' used are presented in figure 6. It can be seen that too much information is —as expected— quite unhelpful. The corresponding notion of recall (again, a somewhat unusual one) provides the anticipated result that the more information, the more likely only the exact solution will be retrieved — see figure 7.

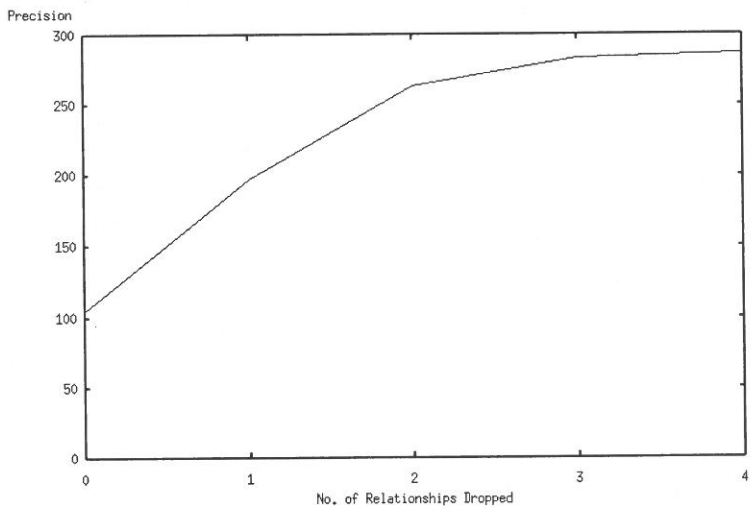

Fig. 6. A Measure of Precision

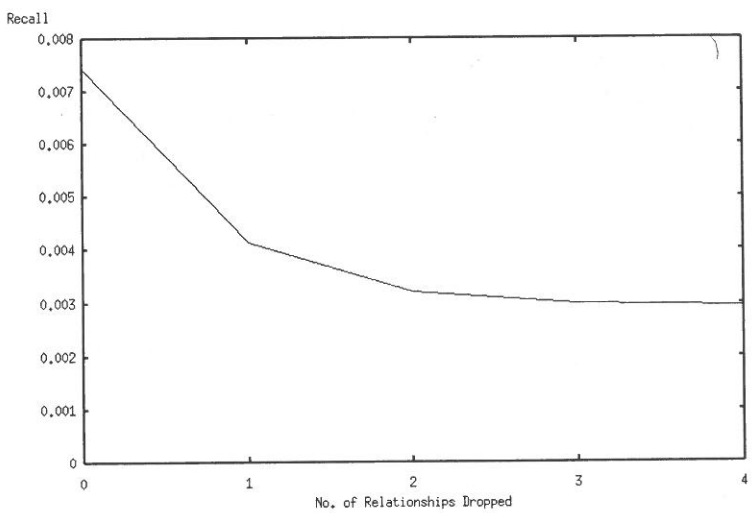

Fig. 7 A Measure of Recall

6. Discussion

Issues connected with the data are discussed with a view to developing more useful retrieval indices.

6.1. What is the Significance of the Data?

First, it should be mentioned that the measures selected are not necessarily the best ones for showing how useful the retrieved cases are. The focus was on whether the correct case (i.e. the code which corresponded to the task description) could be retrieved. In a pedagogical situation involving SunTed, perfect retrieval would not be expected: instead, it would be expected that a similar case would be sought which could be adapted by the student to produce the required solution. Therefore, a more complex measure could have been developed for the precision of the search. This could take into account the 'quality' of each case retrieved.

Despite some doubts about the quality of the data, the evidence suggests that *for retrieval* the benefits of a very simple intermediate description language are worthwhile: easy to learn, and reasonably effective in producing a set of useful cases. Again, for *retrieval* it would appear that a more complex intermediate language is harder to learn and apply, and also less effective at retrieving useful cases. As to the benefits of a complex intermediate language in general, the benefits here are likely to lie with the *use* of a selected case. Constructing a set of constraints on the solution can be seen as a form of code design. It is therefore reasonable to expect the use of the more complex intermediate description language to be valuable. However, the study detailed here does not address this issue.

6.2. What is the Value of Task Descriptions?

It is widely accepted that programmers utilise some form of case-based reasoning when producing a program that satisfies a specification. For novice programmers, such reasoning draws on the relatively limited experience that they have about the programs found as examples in text books (and elsewhere) as well as the programs that a novice has written. It is arguable as to whether or not it is the experience of the task rather than the resulting programs that is the basis for retrieval.

Task descriptions are an important part of learning to program. Robertson and Kahney have stressed the role of "imitation" in learning to program [Robertson & Kahney, 1993]. For them, imitation is a form of analogical reasoning which is based on surface features. These surface features need to be drawn from both task descriptions and programming solutions. Robertson and Kahney stress the role and importance of example tasks (specifications). Chi and her colleagues have both demonstrated the different ways in which novices and experts classify tasks, and indicated the significance of the 'so-called' self-explanation effect [Chi *et al*, 1989] which is based on the use of both task descriptions and their worked solutions.

So if tasks are so important, why not incorporate them in the indexing system? Part of the reason is the difficulty of encoding the necessary information. Mostly, tasks are presented as informal (and incomplete) specifications, expressed in relatively unconstrained natural language. The entities are much more varied than the set of entities introduced to the novice programmer. Relationships between elements in the task domain can be expressed in many ways.

Over the last few years, Weber has produced some impressive results for the automated selection of the best case (based on remindings) [Weber, 1995, Weber, 1996], but this is at the cost of producing quite elaborate task analyses for each program. SunTed's advantage is that the information currently used to retrieve cases is relatively quick to generate.

In the future, SunTed could be used as the basis for the kind of case-based retrieval that indexes the library of tasks. The task description language has to be developed, and ways found of applying it efficiently to build a library of cases.

6.3. Intermediate Representations

The development of an incremental approach based around a series of intermediate description languages is related to the development of expertise in physics as described by White [White, 1993]. In White and Frederiksen's

PQUEST system, students encountered a number of kinds of model of an electric circuit. The various models were not those necessarily held by experts but they went some way to providing intermediate models of electricity.

The same can be done — to an extent — for learning to program. However, the programming community has not really developed widely accepted intermediate representations. Even within the Prolog community there are many different ways of describing the structure of program code. For example, Richard O'Keefe uses a mathematical formalism to describe the code in his "Craft of Prolog" [O'Keefe, 1990].

Experts typically have different ways of representing knowledge about programs and programming techniques. Top level experts often make use of very abstract descriptions of Prolog Programming Techniques. Some appeal to abstract mathematical theory and may make use of complex and highly abstract operations to manipulate techniques in suitable ways. As such representations are seen as expert-like, there have been a number of attempts to explicitly teach programming plans [Soloway, 1986], programming techniques [Brna, 1993, Ormerod & Ball, 1996] and even design patterns [East *et al*, 1996].

Arguably, students need a simplified *non-expert* intermediate description language to provide the scaffolding that will help them learn to become more proficient. The Prolog Programming Techniques utilised in SunTed have this kind of role: these techniques are not necessarily the ones the students will 'end up' with, but they can be expected to help students develop coding skills.

SunTed, the version of Ted utilised in this study, is not designed to *make* the novice programmer access techniques by name. Rather, the environment is designed to allow the novice to exploit the *form* of a Prolog Programming Technique once a suitable case has been retrieved via the intermediate description language provided. This is a novel approach to supporting students though as yet there are no results on the effectiveness of this approach.

There is a price to be paid: a programmer learning a new language —whether Prolog or not — may be an experienced programmer in another language or a complete programming novice. Either way, the first few hours of practical programming are usually difficult: the novice doesn't remember the syntax very well; he/she has problems understanding how the language works — see du Boulay and Monk's notional machine [duBoulay *et al*, 1981]; the novice has a major problem understanding the relationship between the structure of a program and its function; and it is hard for the novice to express his/her intentions in the target language —even when the way to solve a programming exercise is well understood at the algorithmic level.

So, won't an intermediate description language for case-based retrieval just complicate things? The evidence that has been produced indicates that simple constraints such as those given by modes and types are not difficult to use. The explicit use of (Prolog) Programming Techniques for retrieval was not considered — primarily because it was assumed that novices would not find it easy to describe programs in terms of techniques.

As to the description of the constraints that must hold between the arguments of a predicate, the question was whether or not novices could sensibly describe the anticipated nature of the solution to a simple programming task using a specially designed description language.

Some evidence has been provided suggesting that specifying this class of constraints is relatively unrewarding for the purposes of *retrieval* only — though this is dependent, to an extent, on the case-based retrieval method used by SunTed. On an anecdotal level, subjects did not find the task easy. However there is a persuasive argument for this form of intermediate description language in terms of the value it has for helping students to think about how a retrieved case can be exploited. This issues has not yet been explored but it can be anticipated that the effort in learning to use this kind of description language will have long term benefits.

7. Conclusion: Learning Opportunities Revisited

If learning opportunities can be identified, then their exploitation requires that novices are not automatically prevented from engaging in problem solving. It is tempting, given an effective

procedure for locating the 'best' case, to offer the novice this best case straight away. Weber can do this for a set of LISP programming tasks [Weber, 1995, Weber, 1996]. Doing so removes an opportunity for the novice to develop improved intermediate descriptions for retrieval. So a scaffolding tool might support the process of forming intermediate descriptions. Further, such a tool might also provide support for extending the description language — a problem that has not been addressed here.

Support for novices learning to retrieve may sensibly follow a MAC/FAC-like approach [Forbus *et al*, 1995]: searching using somewhat superficial criteria before selecting the best case based on 'deeper' criteria. This is consistent with the use of a simple retrieval language. Given that SunTed meets the basic requirements, extending SunTed to provide coaching for novices might well be very effective.

Other learning opportunities described in section 1 have not been addressed to any great extent. The extension of systems like SunTed to allow for novices to develop their own library of indexed cases would be an advantage. It would be relatively easy to provide facilities for novices to use the supplied intermediate description language for constraints on the solution — but much harder if novices are to be supported in developing accurate indices for tasks. If the system can be designed so that it is not too dependent on requiring accurate indices, then it may well be able to provide a usable system but possibly at the cost of making it harder to retrieve desirable cases.

Support for the application of the case to the current task depends to an extent on providing tools for extracting the useful structure from the case. SunTed provides an interesting approach to this, and does provide the necessary foundation. It is necessary to consider further how such facilities can be utilised to promote increasingly effective application.

Acknowledgements Thanks to Judith Good for her detailed comments, and to Andy Bowles for the programming of SunTed and MacTed. The design of MacTed and SunTed is the responsibility of Andy Bowles, Paul Brna, David Robertson and Helen Pain. The work was funded by the Joint Council Initiative in Cognitive Science and Human-Computer Interaction Grant number G9030396.

References

[Bowles & Brna, 1993] A, BOWLES, P. BRNA, Programming plans and programming techniques, In *Artificial Intelligence in Education, 1993: Proceedings of AI–ED 93*, (P. Brna, S. Ohlsson, H. Pain, Eds.), (1993), pages 788–795, AACE, Virginia.

[Brna, 1993] P. BRNA, Teaching prolog techniques, *Computers & Education*, 20,1, (1993), 111–117.

[Brna *et al*, 1991] P. BRNA, A. BUNDY, T. DODD, M. EISENSTADT, C. K. LOOI, H. PAIN, D. ROBERTSON, B. SMITH, M. VAN SOMEREN, Prolog programming techniques, *Instructional Science*, 20, 2/3, (1993), 111–133.

[Chi *et al*, 1981] M. T. H. CHI, P. J. FELTOVICH, R. GLASER, Categorization and representation of physics problems by experts and novices, *Cognitive Science*, 5, (1981), 121–152.

[Chi *et al*, 1989] M. T. CHI, M. BASSOK, M. W. LEWIS, P. REIMANN, R. GLASER, Self explanations: How students study and use examples in learning to solve problems, *Cognitive Science*, 13, (1989), 145–182.

[duBoulay *et al*, 1981] J. B. H. DU BOULAY, T. O'SHEA, J. MONK, The black box inside the glass box: Presenting computing concepts to novices, *International Journal of Man Machine Studies*, 14, (1981), 237–249.

[East *et al*, 1996] J. P. EAST, S. R. THOMAS, E. WALLINGFORD, W. BECK, J. DRAKE, Pattern-based programming instruction, In *Proceedings of the 1996 ASEE Annual Conference and Exposition*, (1996), Washington, D.C.

[Forbus *et al*, 1995] K. D. FORBUS, D. GENTNER, K. LAW, MAC/FAC: A model of similarity based reasoning, *Cognitive Science*, 19, 2, (1995), 144–205.

[Gamma *et al*, 1994] E. GAMMA, R. HELM, R. JOHNSON, J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

[Linn *et al*, 1992] M. C. LINN, M. KATZ, M. J. CLANCY, M. RECKER, How do Lisp programmers draw on previous experience to solve novel problems?, In *Computer-based Learning Environments and Problem Solving*, (E. Decorte, M. C. Linn, H. Mandl, L. Verschaffel, Eds.),(1992), pages 67–101. Springer-Verlag, Berlin.

[O'Keefe, 1990] R. A. O'KEEFE, *The Craft of Prolog*, MIT Press, 1990.

[Ormerod & Ball, 1996] T. C. ORMEROD, L. J. BALL, An empirical evaluation of TEd, a techniques editor for prolog programming, In *Empirical Studies of Programmers: Sixth Workshop*, (W. D. Gray, D. A. Boehm–David Eds.), (1996), pages 147–161, Ablex Publishing Corporation, New Jersey.

[Pennington, 1987] N. PENNINGTON, Comprehension Strategies in Programming, In *Empirical Studies of Programmers: Second Workshop*, (G. M. Olson, S. Sheppard, E. Soloway Eds.) (1987), pages 100–113, Ablex Publishing Corporation, New Jersey.

[Reimann *et al*, 1993] P. REIMANN, S. WICHMANN, T. J. SCHULT, A learning strategy model for worked out examples, In *Artificial Intelligence in Education, 1993: Proceedings of AI–ED 93*, (P. Brna, S. Ohlsson, H. Pain, Eds.), (1993), pages 290–297, AACE, Virginia.

[Robertson & Kahney, 1993] S. I. ROBERTSON, H. KAHNEY, Is analogical problem solving always analogical?: The case for imitation, HCRL Technical Report 97, HCRL, The Open University, 1993.

[Ross, 1987] B. H. ROSS, This is like that: the use of earlier problems and the separation of similarity effects, *Journal of Experimental Psychology: Learning, Memory and Instruction*, 13, (1987), 629–639.

[Schult & Reimann, 1995] T. J. SCHULT, P. REIMANN, Dynamic case-based reasoning, In *Artificial Intelligence in Education, 1995: Proceedings of AI-ED'95 — 7th World Conference on Artificial Intelligence in Education*, (J. Greer, Ed.), (1995), pages 154–161, AACE, Charlottesville, VA.

[Soloway, 1986] E. SOLOWAY, Learning to Program = Learning to Construct Mechanisms and Explanations, *Communications of the ACM*, 29,9, (1986), 850–858.

[Weber, 1995] G. WEBER, Providing examples and individual remindings in an intelligent programming environment, In *Artificial Intelligence in Education, 1995: Proceedings of AI-ED'95 — 7th World Conference on Artificial Intelligence in Education*, (J. Greer, Ed.), (1995), pages 477–484, AACE, Charlottesville, VA.

[Weber, 1996] G. WEBER, Individual selection of examples in an intelligent learning environment, *Journal of Artificial Intelligence in Education*, 7,1, (1996), 3–31.

[White, 1993] B. J. WHITE, Intermediate abstractions and causal models: A microworld-based approach to science education, In *Artificial Intelligence in Education, 1993: Proceedings of AI–ED 93*, (P. Brna, S. Ohlsson, H. Pain, Eds.), (1993), pages 26–33, AACE, Virginia.

A. The Ten Problems

Note that problem 1 and 7 are different wordings for the same task.

1. Write a predicate *delete* which takes a list in its first argument, and an atom in its second, and outputs a list which is the same as the first but with that atom removed. The elements in the output list should be in the same order as in the input list.
2. Write a predicate *len* which takes a list in its first argument, and returns the length of the list in its second.
3. Write a predicate *replace* which takes a list and two atoms and returns a list the same as the input, except all occurrences of the first atom have been replaced by the second.
4. Write a predicate *sum* which takes a list of numbers in its first argument, and finds the sum of those numbers in its second.
5. Write a predicate *doublenum* which takes a list of numbers and a single number and returns a list of numbers. The output list is the same length as the input list, and contains elements which are double the corresponding input element if they are greater than the input number, otherwise they are the same.
6. Write a predicate *intersection* which takes two lists of numbers and returns a single list of numbers such that each element of the answer is found in each of the input lists.
7. Write a predicate *delete* which takes a list in its first argument and an atom in its second, and outputs a list which is the same as the first but with that atom removed.
8. Write a predicate *squarelist* takes a list of numbers and returns a list of the squares of those numbers.
9. Write a predicate *alessnum* which tests element by element whether a list of numbers is numerically less than another list.
10. Write a predicate *position* which returns the position of a given element in a given list of atoms.

Received: September, 1997
Accepted: January, 1998

Contact address:
Paul Brna
Computer Based Learning Unit
Leeds University
LEEDS LS2 9JT
England UK
phone: +44 113 233 4637
fax: +44 113 233 4635
email: paul@cbl.leeds.ac.uk

PAUL BRNA Paul Brna is a Lecturer in the Computer Based Learning Unit, the University of Leeds. He has worked on the applications of AI to Education, including research into the interpretation and use of external representations, and learning environments for program construction and debugging. He has authored over eighty technical papers.