Learning About Programming

GORDON I. MCCALLA AND JIM E. GREER
ARIES Laboratory
Department of Computational Science
University of Saskatchewan
Saskatoon, Saskatchewan S7N 0W0, CANADA

Abstract: For many years in the ARIES Laboratory we have been involved in building tools to support the learning of programming. In this paper several of these projects are discussed. One of these projects, SCENT, has investigated how to give strategy-level advice to learners on their solutions to LISP recursive programming problems. The SCENT advisor must do deep diagnosis of learners' strategies to essentially understand the underlying logic of their solution. Another project has resulted in a system called PETAL that scaffolds learners' deep problem solving as they create solutions to LISP programming problems. PETAL does not do any deep diagnosis, but has been designed based on empirical evidence as to how learners think about problem solving at the mental model level. Recently, the PETAL scaffolding environment has been attached as a front end to the SCENT advising system to provide a symbiotic system that can be used in a distance learning context. Learners at a distance use the relatively inexpensive PETAL system and when they need advice on their completed solutions they can electronically contact a centrally running version of the relatively expensive SCENT advisor for help. A final ARIES project, the G.E.N.I.U.S. PL/C programming advisor, does an end run around the need for deep knowledge (either internalized or externalized) by posing as an expert and thereby helping learners to help themselves. No deep knowledge is incorporated into G.E.N.I.U.S. (in fact it is "ignorance based" rather than knowledge based), but it does achieve robust, albeit often flawed, performance.

Keywords: recursive problem solving, knowledge-based advice, scaffolding environments, ignorance-based reasoning, distance learning, artificial intelligence in education, individualized support for learning

Introduction

Programming is a complex activity, requiring a variety of problem solving skills. A programmer is required to formulate a strategy for solving a programming task, transform this strategy into a highly restricted programming language, and then reformulate either the strategy or re-consider implementation decisions as execution of the program reveals flaws in design or implementation. A variety of techniques are used in carrying out this difficult activity. For example, deep mental models of how programs work or of how abstract machines execute programs may be used in strategy creation or debugging; the programmer may resort to modifying previously successful solutions to related tasks; or he or she may instantiate or adapt templates outlining typical solutions to generic tasks. Learning how to program is therefore no trivial matter; and neither is it easy to create tools to support the learning of programming.

For many years in the ARIES Laboratory we have been involved in building such tools. We have also carried out empirical studies of aspects of how programmers solve problems as well as studying the effects of the tools we have built on the novice programmers who have used them. Our long term goal goal is to provide novice programmers with learning environments that are adaptable to the specifc individual needs of each novice.

There are two basic approaches to individualization in the intelligent tutoring systems (ITS) community. In one appoach, ITS researchers have explored a variety of discovery learning environments (White and Fredericksen, 1985; Smith, 1986) that provide an enriched microworld in which a learner can explore concepts from the domain of study through creative play. Within the constraints of the discovery environment, the goals of this "creative play" are chosen by the learner, to satisfy his or her individual objectives. Some discovery environments help reduce the complexity of the learning process by providing "scaffolding" for various concepts early in the learning experience (Soloway et al., 1993; Bhuiyan, 1992). Scaffolding can slowly fade away as no longer needed, normally at the discretion of the learner. Individualization in a discovery environment is thus mostly self-actuated by the learner.

In contrast, the other approach to individualization assumes the system can take a more proactive role in crafting the learning environment to suit a specific learner. Using student modelling techniques (Self, 1994; McCalla, 1992; Greer and McCalla, 1994), the system tries to track aspects of the learner's understanding of the domain and to adjust the learning situation according to the perceived cognitive needs of the learner. While this approach may seem more dictatorial and the learning environment may seem less inspirational to the learner than is a self-actuated discovery environment, this is not necessarily so. In fact, for truly effective discovery learning, it is very useful to be able to direct the learner when he or she gets stuck on a learning plateau or to be able to automatically reduce scaffolding as the system detects that it is no longer needed. This requires the system to diagnose important aspects of a learner's cognitive state, i.e. it requires student modelling. Moreover, really good student modelling can be almost invisible to the learner and not seem constraining at all.

In our work in the programming domain we have taken both approaches to individualization. Our work on the SCENT project (McCalla et al., 1988) takes the second approach: it is aimed at providing robust and subtle student modelling techniques that can be used to "understand" the deep logic (what we call the "strategy") underlying a learner's solution to a LISP (or Scheme) recursive programming task. The goal is to be able to diagnose logical errors in learners' completed solutions to programming tasks and to provide advice in English to the learners about their solutions. But, in line with the self-actualization we want to achieve, this advice is provided to the learner to be consulted at his or her discretion. The SCENT advisor is just that: an advisor not a controller.

Another ARIES project, PETAL (Bhuiyan, 1992) takes the first approach: providing a scaffolding environment to support the learning of LISP (or Scheme) recursion. PETAL delves deeply into the probelm solving process, directly supporting learners in their deep "mental model" level problem solving as they create solutions to programming tasks. While SCENT critiques the logic underlying completed solutions, PETAL provides support during the creation of the solution. Prototype PETAL and SCENT systems have both been implemented and tested with real learners (McCalla and Greer, 1993).

Recently, we have put the PETAL system and the SCENT advisor together in a distance learning situation, and tested the symbiotic system with real learners (Price et al, 1994). PETAL, which can run on relatively inexpensive Macintosh machines, is provided to learners to use "in situ". From PETAL learners can be linked electronically to SCENT, which must be situated centrally since it is computationally intensive and must run on expensive high powered workstations. Learners thus receive from PETAL the support they need locally as they develop solutions to their programming tasks, but are also able to receive from SCENT individualized advice about their solutions when they produce a complete solution, or when they reach an impasse. This we believe to be a compromise that allows systems from two different streams of

AI-Ed research to work synergistically to provide a truly useful learning environment in a cost-effective way.

In a fourth project, we have done an "end run" around the need to support the learner's problem solving at all. Instead, we have looked at the relationship between the advisor and the learner and have taken advantage of idiosyncracies in the social roles assumed by novices and (supposed) experts. In particular we have built a PL/C advising system called G.E.N.I.U.S. (McCalla and Murtagh, 1991). Although G.E.N.I.U.S. is almost completely knowledge free, it poses as an expert programming advisor. As in the ELIZA system (Weizenbaum, 1966), the illusion of experthood can, at least occasionally, be real enough that the learner is encouraged (by feedback from G.E.N.I.U.S.) to continue to work on a programming problem and solve it on his or her own. This "ignorance-based" approach demonstrates the possibility of using the social context underlying the learning situation in order to provide effective assistance without needing to do deep cognitive modelling at all.

In this chapter, both SCENT and PETAL are briefly synopsized, and we discuss how the two systems can be linked. Then, the G.E.N.I.U.S. approach is discussed as a possible alternative to deep, knowledge-based approaches. The chapter concludes with some of the lessons we have learned from these various research projects about helping novices learn programming, and with some suggestions for future research. We believe that many of these lessons apply widely beyond the programming domain.

The SCENT Project

The goal of the SCENT project has been to develop an intelligent advising system for LISP programmers (McCalla et al., 1988). A novice programmer presents to the SCENT advisor a recursive program to solve a programming task and the advisor replies with strategy-level explanations about aspects of the learner's program. The novice can browse through these explanations at many levels of detail in order to gain understanding of what he or she has done appropriately or inappropriately.

The SCENT advisor takes a standard "diagnosis" approach to individualization in that it builds an internalized model of the student programmer's strategy in order to comment upon it. There are many examples of such model-based recognition schemes in the programming domain, for example (Johnson and Soloway, 1984; Wills, 1990). In our approach we attempt to overcome some of the problems of model-based recognition, especially problems in robustness, flexibility, and adaptability by using "case-based" and "granularity-based" reasoning methodologies.

Our granularity-based reasoning methodologies extend some of the ideas of Hobbs (1985). We characterize a domain in terms of models arrayed at various levels in two orthogonal dimensions: abstraction (general/specific) and aggregation (whole/part). The resulting graph is called a "generic granularity hierarchy" for the domain. Each model relates to other models with respect to their level of detail (granularity) along both abstraction and aggregation dimensions. In cognitive domains, like the SCENT recursion domain, these models correspond to concepts potentially used by people solving problems in the domain. The existence of particular concepts is recognized ("diagnosed") from observed behaviour by an inference system (the so-called "recognition engine") associated with the generic granularity hierarchy. Organizing knowledge into granularity hierarchies improves robustness in recognition, since even if concepts cannot be recognized at a fine grain size, they may be recognized at coarser grain sizes. Concepts recognized (or partially recognized) based on learner behaviour can be used to characterize a learner's problem-solving strategies. Our granularity-based diagnosis approach is more fully elaborated in McCalla et al. (1992).

After doing granularity-based diagnosis, SCENT consults a library of cases in an attempt to find a previous solution to the programming task that is strategically similar to the one just created by the learner. In retrieving a case, SCENT does not match the program code or even the strategies themselves, but instead retrieves a case which matches the pattern of strategic

instantiations that were found to exist in the learner's solution. The learner's instantiated hierarchy is compared to the stored case hierarchies using efficient algorithms developed by Coulman (1991) to find the most similar matching case. Variations and deviations between the learner's strategies and the best-matching case form the basis for advice generation. Furthermore, annotations pre-stored with the retrieved matching case can also be provided as advice (at varying grain sizes) for the learner. Case-based reasoning further enhances robustness by removing the necessity to anticipate all possible learner behaviours. Instead, actual learner behaviour on a task can be slowly accumulated as typical cases representing this behaviour are discovered and stored in the case library. This alleviates many of the problems normally associated with incompleteness in model-based diagnosis. Together, granularity and case-based reasoning help the SCENT advisor to overcome the usual brittleness inherent in traditional model-based diagnosis.

In a typical session with the SCENT advisor, the learner creates a program which is intended to solve one of the many programming problems known by SCENT and then requests advice on the solution. Using the combined granularity/case-based reasoning process sketched above, SCENT does a strategic analysis of the learner's code, finds the closest matching stored case, generates natural language expressions describing differences (at many levels of detail) between this stored case and the learner's approach, and then makes both the expressions representing the differences and the pre-stored explanations associated with the retrieved case (also arrayed at many levels of detail) available to the learner. SCENT can literally recognize the presence or absence of hundreds of strategic elements in the learner's program, such as the type of recursion, the suitability of the base cases, the intended reduction step, the composition of the returned result, etc. This means that the advice that is available can be copious and hard to understand.

Fortunately, the fact that the advice is multi-layered along both aggregation and abstraction dimensions provides underlying structure for learners to browse, thus simplifying their task in understanding the advice. Learners access this multi-layered advice by selecting (through mouse clicks) the part of their code about which they want advice. (This is a decision about the desired aggregation level of the advice.) SCENT can also allow the learner to select the level of detail of the advice itself or it can make this choice for the learner. (This is a decision about the desired abstraction level of the advice.) Currently, SCENT does the initial selection of the abstraction level of advice at what it perceives to be the most relevant level of detail. The advice is scrolled into an advice window as the different pieces of code are highlighted.

SCENT is capable of recognizing strategies in a wide range of learner programs, even when the code strays significantly from correct solutions. SCENT also generates useful advice, pointing out where the learner's program is flawed and giving hints as to how it can be improved. This powerful recognition and advising capability does not come without a price. In order to achieve reasonable real-time performance, SCENT requires a considerable amount of computational power and memory. It is also important for the effective functioning of SCENT that learners create syntactically correct code, even if it is conceptually flawed. These are serious impediments to making the SCENT advisor available to a larger number of learners. Later in this chapter we will discuss an experiment in trying to make the SCENT advisor more widely available by end-running the performance problem and the correct syntax restriction.

We also would like to make the SCENT approach to granularity/case-based reasoning widely available in other domains. We strongly believe the SCENT approach transcends the LISP domain, in fact generalizing beyond programming to other problem solving domains. However, it is not particularly easy in any new domain to craft the generic hierarchy and to stock the case library with typical solutions. To overcome this impediment to making the SCENT approach itself widely available, we have been building knowledge engineering tools to support the SCENT approach. These tools are collectively called AROMA.

There are two main tools in AROMA: the hierarchy engineering tool, supporting the creation of the generic hierarchy; and the task engineering tool to help the knowledge engineer augment the case library to handle a new task. The hierarchy engineering tool provides support for creating strategy objects, abstraction and aggregation linkages, contexts and constraints (used by the

recognition engine), and generic advice that can be "inherited" by cases. The knowledge engineer can browse through a graphical version of the generic hierarchy to help him or her visualize the design as it grows. The recognition engine is available so that the generic hierarchy's ability to "understand" real world behaviour in the new domain can be tested. Also, conceptual-level debugging capabilities and other augmentations of the hierarchy engineering tool are now being added. Figure 1 shows a typical hierarchy engineering screen.

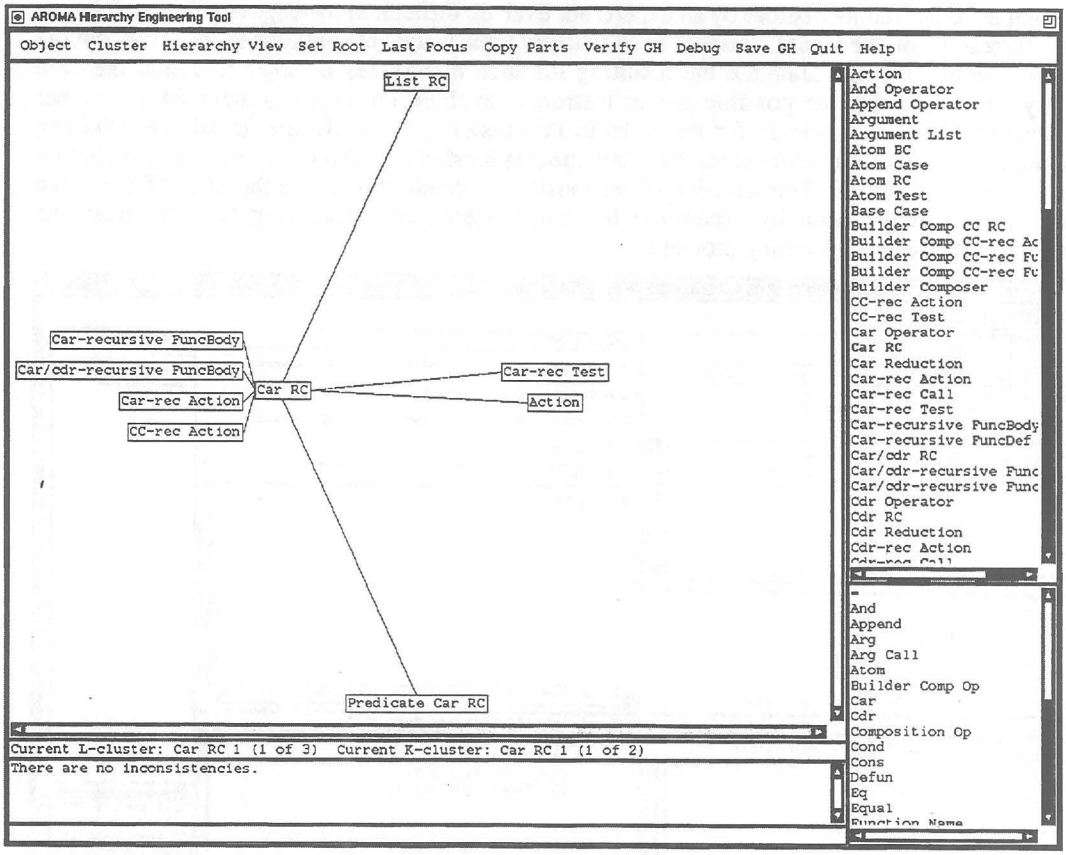

Figure 1. - A Hierarchy Engineering Screen from AROMA

The task engineering tool allows the knowledge engineer to access both the generic hierarchy and the recognition engine. The recognition engine is used for any task in order to create the instantiated version of the generic hierarchy that constitutes the basis for a case. The task engineering tool also provides special capabilities to help the knowledge engineer annotate cases with case-specific advice and modify advice "inherited" from the generic hierarchy. Visualization tools to view the instance hierarchy are also available. Figure 2 shows a typical task engineering screen.

Implemented in Allegro Common LISP, AROMA is currently nearing "beta" status. We have used the "alpha" version extensively in student projects and other small test domains in our laboratory. Once AROMA is fully operational, our hope is that it will be possible to create a generic hierarchy in a matter of days or weeks (depending on the complexity of the domain and the amount of codification already done in the domain). Given an existing generic hierarchy for a domain, we feel that the task engineering tool will make it possible for a knowledge engineer to create a case library for a new task in that domain in a matter of hours. If these performance

objectives can be met, then it should be possible to use the SCENT approach in realistic settings. For example, it would be possible to create an on-line advising system for critical portions of a university or high school course by having a domain expert develop a generic hierarchy (or hierarchies) for these portions during the summer before the course is offered. Then, the course instructor could cobble together a case library the night before an assignment containing tasks that test this critical material is handed out. The instructor would not have to build a generic hierarchy (it being done the summer before by an expert) nor even do difficult knowledge acquisition for each task. Instead he or she would "merely" need to know typical student solutions to the tasks, run the recognition engine, and annotate the resulting instance hierarchies in order to create the case library. It would thus be possible for an instructor to think up an assignment for his or her students to work on, to create for the tasks in this assignment an on-line, intelligent advising system, and to do so the night before the assignment is handed out. No more students showing up at all hours for advice! This division of responsibility should thus make the SCENT approach much more widely useful by separating the tough hierarchy engineering problem from the relatively easier task engineering problem.

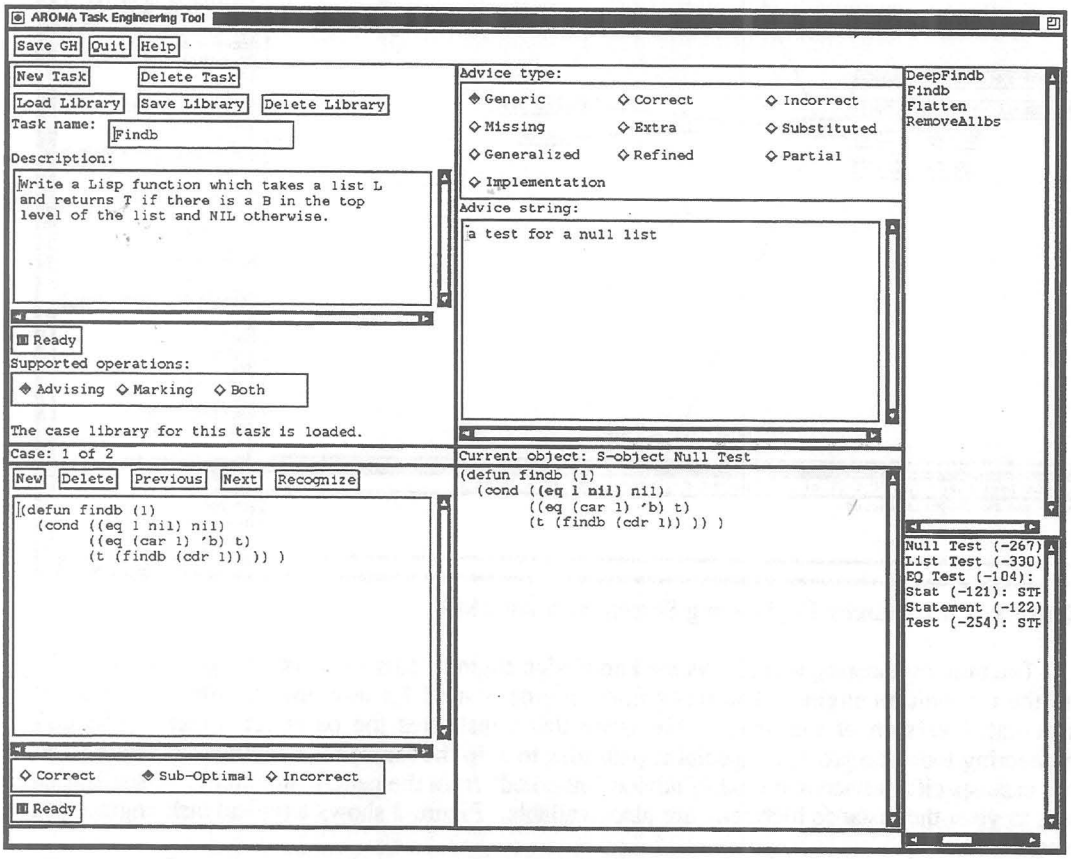

Figure 2. - A Task Engineering Screen from AROMA

The PETAL System

The SCENT advisor provides advice at the strategy level, but this won't help learners who are having problems formulating their strategies in the first place. Their mental model level may be flawed. However, since mental models of programming do not correspond in any obvious way to

the real world physical models usually referred to in the mental modelling literature (see Gentner and Stevens (1983)), we use the term "mental method" to capture the idea that the models represent complete problem solving strategies.

In our attempts to study the mental methods used by LISP programmers, numerous protocol studies have been conducted (Bhuiyan, 1992; Bhuiyan et al, 1991; Bhuiyan et al., 1989). We discovered that in addition to flawed mental methods of recursion (c.f. Kahney, 1982; Kurland and Pea, 1985), there are also a number of correct mental methods of recursion at varying levels of sophistication. One of these is the *syntactic method*, where learners abstract structural features and code templates in order to put together their solutions. A more sophisticated problem solving approach is the *analytic method*, where learners no longer see recursion in terms of syntactic templates, but instead analyze the problem in terms of cases, and determine input conditions and output strategies for each case. Perhaps the most sophisticated mental method of recursion is the *analysis/synthesis method*, where learners are able to break a problem down into sub-problems and then combine the sub-problem solutions appropriately. Empirical studies carried out in the ARIES laboratory (Bhuiyan et al., 1989; 1991) have shown that learners indeed do use syntactic and analytic methods, and that there is a natural progression from syntactic to analytic methods as students gain experience in writing recursive programs. We hypothesize that as learners become experts, they will make a final transition to the analysis/synthesis method, although the studies only ran long enough to hint at this evolution.

Shawkat Bhuiyan devised a scaffolding system called PETAL (Bhuiyan, 1992) which consists of three Programming Environment Tools (PETs) to support learners in the use of the three mental methods of recursion: PET1 for the syntactic method, PET2 for the analytic method, and PET3 for the analysis/synthesis method. As learners use a PET, they express mental model-level decisions by actions carried out in the PET. These actions can be automatically recorded, and when correlated with external observations made by an experimenter, can yield detailed insights into the use of mental methods. PETAL can also generate runnable LISP (or Scheme) code from the mental-model level descriptions the learners create when using a PET. This helps learners to offload onto PETAL the necessity for dealing with the intricacies of language syntax and semantics, thus allowing them to concentrate on problem solving rather than coding.

In PET1 the learner goes through two stages: stage I where a recursion template is constructed by the learner; and stage II where this template is fleshed out by replacing each step with fragments of code that carry out the step, the fragments also selected from a menu. The learner is urged to finish stage I before proceeding to stage II, but may in fact go back and forth as he or she sees fit. When satisfied with the stage II solution, the learner can ask PET1 to generate program code that ties together the steps of the stage II solution in a syntactically correct way. PET1 thus encourages the learner to structure the problem solving process and to think about designing a solution in terms of existing syntactic templates, hence its name: syntactic PET. Figure 3 shows a typical PET1 screen and Figure 4 shows PETAL's code generation capabilities.

In PET2 the learner goes through three stages in designing a solution to a programming task: stage I, the intention stage, where from a menu the learner selects natural language phrases to describe typical input cases and corresponding output strategies for each case; stage II, the plan stage, where input conditions and output actions corresponding to the input cases and output strategies are selected; and stage III, code generation, where code chunks are chosen to replace the natural language conditions and actions generated in stage II. As in PET1, PET2 will take this stage 3 solution and turn it into a complete, syntactically correct LISP function when the learner is ready. Also as in PET1 the learner is encouraged, but not required to work through the stages in sequence. Since the learner thinks of his or her solution in terms of typical I/O cases, PET2 does support the analytic mental method. Figure 5 shows a typical PET2 screen.

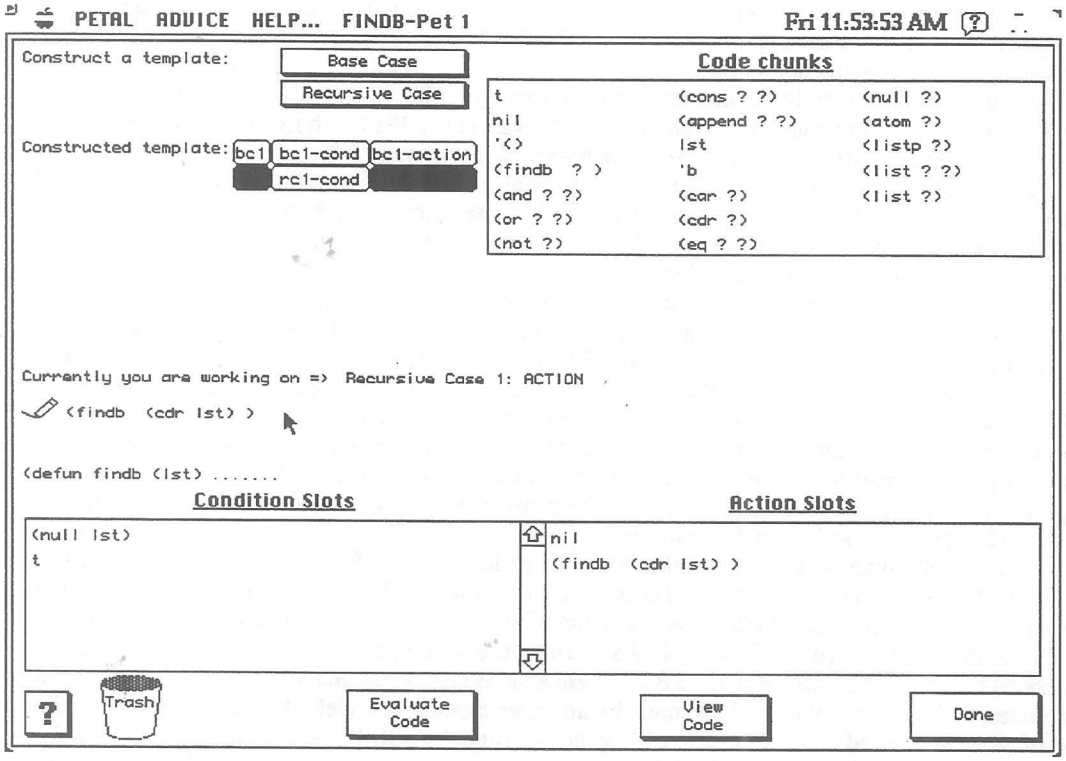

Figure 3. - The PET1 Interface

Program synthesis using the analysis/synthesis PET (PET3) has the following three stages: stage I, where the original problem is broken into subproblems, out of which at least one is the smallest subproblem with a known solution; stage II, where solutions are derived for the reduced subproblems, and the relations among these solutions to the overall solution are determined; and stage III, where the stage II solution is translated into LISP code. Since PET3 has not been extensively tested "in action" with real learners, it will not be further discussed.

PET1 and PET2 have been tested in two different empirical studies (Bhuiyan et al, 1991; Bhuiyan, 1992). In both studies these PETs have proven to be effective learning environments. The Bhuiyan (1992) experiment was a control-group study. The 9 students in the study were divided into two groups: one group of 5 students (the treatment group) using PETAL and the other group of 4 students (the control group) using a normal LISP environment. For both groups a teaching assistant sat with individual learners, taking notes and dispensing occasional advice when learners ran into impasses. During the experiment, the 5 learners using PETAL used both PET1 and PET2 extensively. They attempted an average of 8.8 problems, eventually got 82% of them correct, and took an average of 31 minutes to solve a problem (or to reach an impasse). In contrast, the 4 non-PETAL learners only attempted an average of 4.5 problems each, only got 15% of them correct (the rest were impasses), and took an average of 56 minutes on each problem. The difference in performance between the two groups on the pencil and paper post test was even more startling. The PETAL learners got 60% of the problems completely correct, and 87% of them nearly correct. The non-PETAL learners solved none of the problems correctly and only one solution to one problem was nearly correct. Clearly, the support given by PETAL was crucial in the treatment group learning how to solve recursive problems.

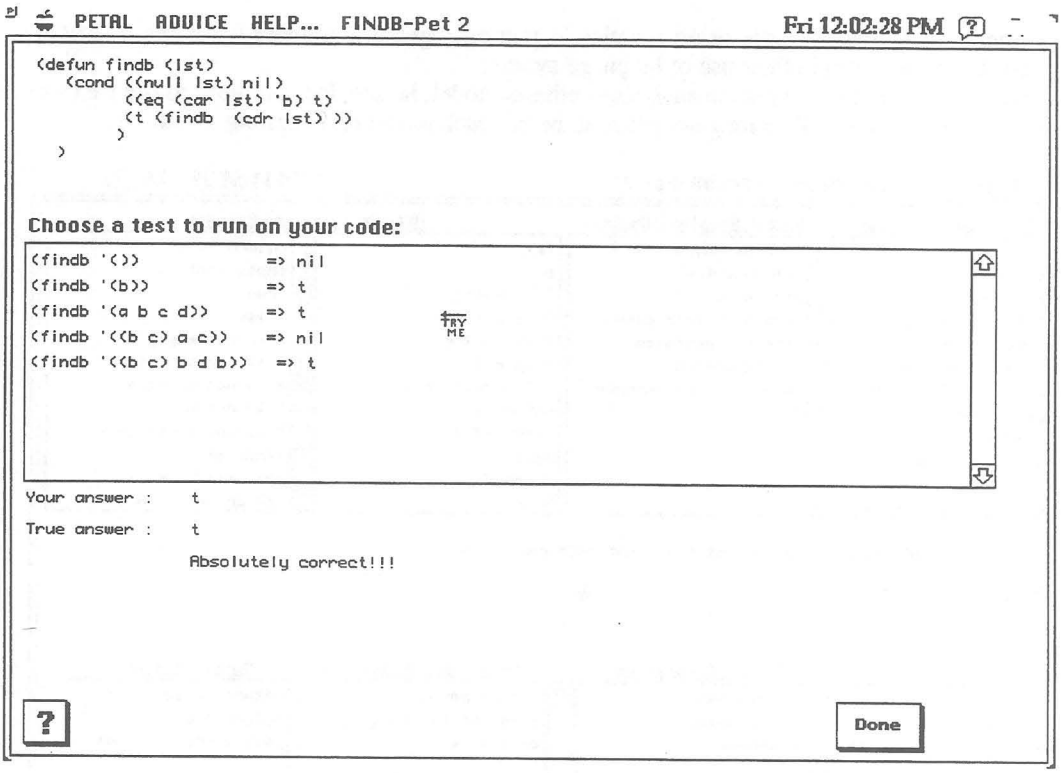

Figure 4. - Code Generation and Testing in PETAL

Our experience with using PETAL have allowed us to make the following observations:

- Certain features of PETs (such as the templates in PET1 or the input conditions/output strategies of PET2 or the division of the problem solving into several learning stages) correspond to deep problem solving methodologies, and hence directly support learning of these methodologies;
- Other features of PETs (such as code generation) relieve the learner from having to be burdened with the surface aspects of problem solving, letting him/her focus on deep levels of cognition;
- Behaviour at the problem solving level is easier to observe from the protocols using PETs than not using them, hence suggesting that in addition to supporting the learner, PETAL may also be a useful tool for the experimenter seeking insight into deep problem solving;
- Revolutionary change occurs at least once; that is learners in the control group suddenly seemed to "catch on" at some point, after which problem solving became both more sophisticated and easier;
- Use of a PET does not guarantee use of a problem solving methodology; for example, some learners in the treatment group were sometimes observed to be thinking out their problem solution in syntactic terms but expressing them in the analytic model;
- Old methodologies seem not to fade away, but continue to be used; learners flip-flop back and forth depending on the particular programming task being solved; selection of a methodology depends on the complexity of the problem (syntactic for simple problems; analytic for a combination for more complex problems);
- Scaffolding is needed less and less as expertise increases; each member of the treatment group was able to "fly on his own" after the revolutionary change occurred, to the point that the

treatment group, despite little or no practice in putting together complete functions, outshone the control group even in their use of language syntax;

- By the time a learner acquires the analysis/synthesis model, he/she does not need to use PETs to support learning; thus PET3 may not prove to be of much use in real learning situations.

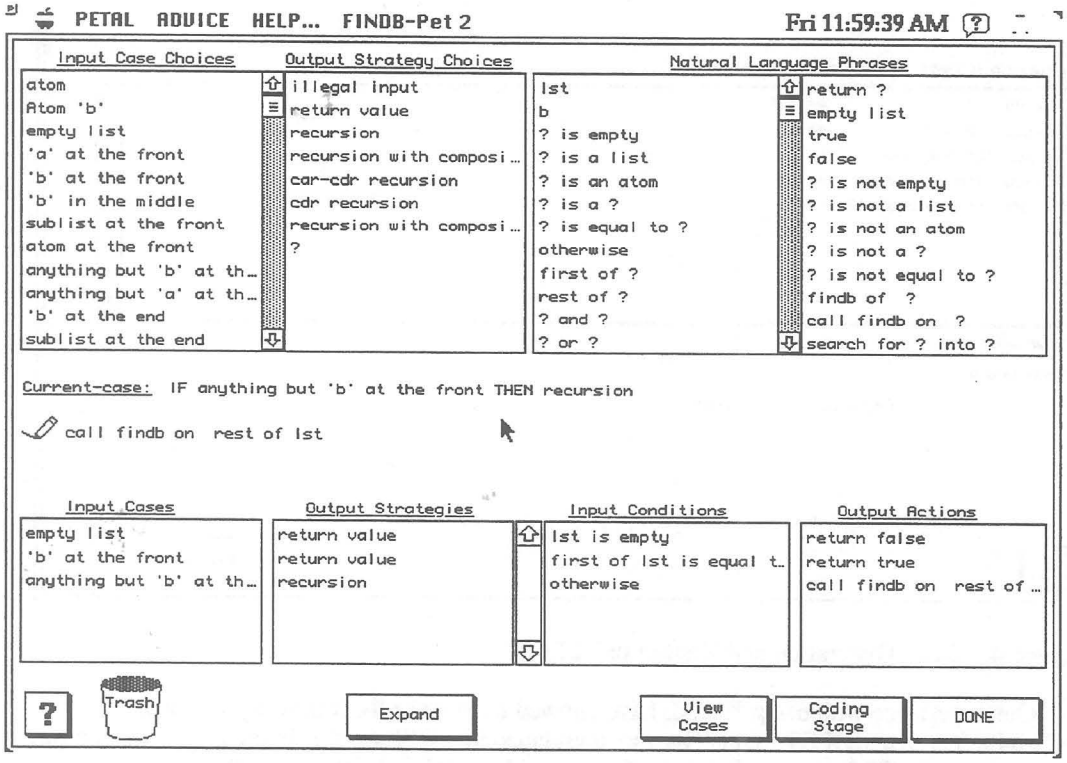

Figure 5. - Intention and I/O stage in PET2

Overall, PETAL has proven to be an effective environment for learning recursion. The PETs provide scaffolding and support for the learners as they carry out programming tasks. The learner is free from dealing with language syntax, and is directly supported at the deep problem solving "mental method" level.

Despite its successes, PETAL suffers from being unable to critique completed programs. This is especially important when a learner reaches an impasse. In our experiments with PETAL, learners were helped over impasses by a human tutor. It seems reasonable to conjecture that adding SCENT's advice giving ability to PETAL may help to ameliorate the need for such human intervention. Moreover, since PETAL runs efficiently on a modestly configured Macintosh computer while SCENT requires a powerful workstation, it seems to be logical to attach multiple, possibly geographically distributed, PETALs to a single, central, SCENT system.

Connecting PETAL and SCENT

The concept of a powerful intelligent tutoring system located centrally but connected to satellite interfaces running on inexpensive personal computers is an attractive model for distance education utilizing ITS technology. There are several ways in which PETAL and SCENT could be integrated to achieve this distributed intelligent tutoring environment. For example, the learner's

LISP code produced in one of PETAL's PETs could be sent via network to SCENT for analysis. Other information that PETAL gathers about mental models and learners' abstract programming plans could also be sent to the SCENT diagnostic engine. The SCENT system could directly communicate with the learner by popping up an advice browsing window on the learner's personal computer. Alternatively, SCENT could send the advice data structure to PETAL over the network and the advice browsing would need to be managed by PETAL.

We chose, for our initial integration attempt, to have PETAL send only the learner's code to SCENT, and after analysis, to have SCENT return the advice data structure to PETAL. Thus it became necessary to enhance PETAL with an advice browsing capability. Figure 6 shows PETAL's advice browser and the advice provided by SCENT corresponding to the highlighted student code.

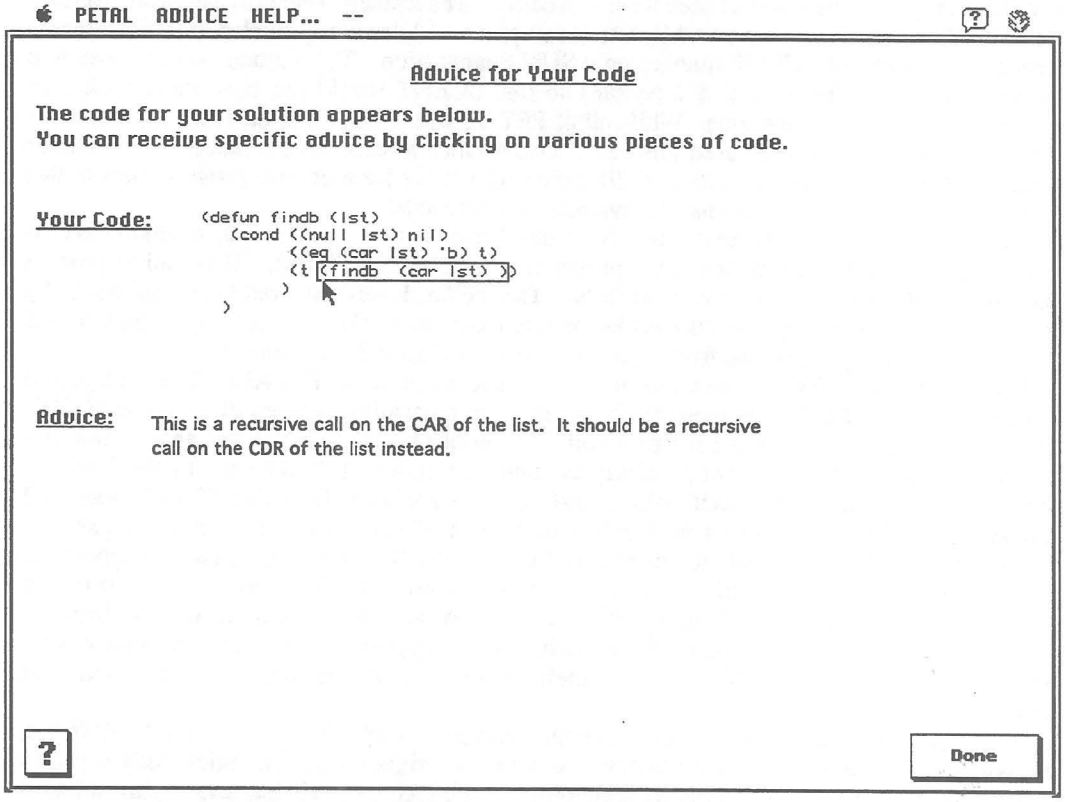

Figure 6. - The PETAL Advice Browser

In order to run multiple PETALs at the same time and possibly in different locations, it was necessary to convert SCENT into an advice server, managing a queue of advice requests and communicating with multiple PETAL clients. The communication protocol involves PETAL sending a request to SCENT, i.e., sending a learner's solution and an indication of the problem being solved. SCENT responds with an estimate of the time that the learner must wait for advice to be returned. When the learner's solution has been analysed and the advice has been generated by SCENT, the learner is informed that advice is ready. Then the learner can browse the advice and SCENT can proceed with handling another advice request.

A pilot study was conducted in the ARIES lab to test the PETAL/SCENT integrated system (see Price et al, 1994). The goals of this study were limited and two-fold: (i) technical and (ii) educational. Our technical goals were to test the ability of the integrated system to work properly and robustly with a distributed set of learners in "real time". Our educational goals were merely to see if the learners who used the integrated system seemed to have a rewarding and fruitful learning experience. We had no desire at this stage to embark on a detailed study, complete with control groups, statistically significant numbers of subjects, post-tests, etc. The critical objective was to test feasibility, not to prove effectiveness.

In the study, 9 learners from an introductory programming class (in which they were learning Scheme) used the PETAL/SCENT system to solve a set of simple recursive programming problems. The subjects ranged from novices who had some programming in Pascal to fairly experienced programmers who had seen recursion before. The learners were provided with a version of the PETAL interface running on a Macintosh computer. Advice was provided to PETAL over a network by a version of SCENT running on a SUN Sparcstation. The learners were broken into three smaller groups (of 4, 3, and 2 people) so that SCENT would not have more than four simultaneous users at any one time. While all of PETAL was available to the learners, they were trained only on PET1 and only used this PET. This seemed a reasonable limitation since earlier protocol studies (Bhuiyan et al, 1991, 1989) indicated that novice recursive programmers at this stage of learning recursion tend to use the syntactic mental model.

Each of the subjects was asked to solve a number of simple problems in recursion and to provide feedback on the quality of support provided by SCENT and PETAL. They had full choice as to which of the several problems to work on. The feedback was gathered in several ways: by human monitors supervising the students in the lab, from protocols automatically logged by the PETAL and SCENT systems, and from interviews conducted after the experiment.

The experiment seems to have met its goals. Technically, the PETAL/SCENT integrated system worked well. PETAL worked efficiently; the communication between PETAL and SCENT also worked; multiple distributed learners could be serviced; and SCENT was able to return a diagnosis within a minute or two; quickly enough that it was still relevant to the learner's problems. Educationally, our exit polling and protocol analysis show that PETAL was well received and the learners found it worthwhile to contact SCENT when stuck at an impasse in problem solving. It seems that together PETAL and SCENT provided constant support for problem solving combined with at least somewhat useful individual guidance on problem solutions when needed. Overall, many of the learners wanted to continue to use the learning environments, even after the end of the experiment, to support their learning as their course proceeded. This is perhaps the most compelling measure of the quality of their learning experience.

While this study is only a "proof of concept" feasibility study that PETAL and SCENT can work symbiotically at a distance, we believe we are on the right track. The pilot study supports the idea of using the simple scaffolding technology of PETAL to guide learners during solution development and taking recourse in a centrally located intelligent tutoring system to help learners over impasses. However, further work needs to be done to follow-up on the feasibility study. In particular we want to study whether the advice can be improved. Currently, the advice provided by SCENT does not take into account the nature of the PET that learners are using. This wasn't a big problem in the experiment, since all learners used PET1 and the advice was based on structural features of the learners' code, perfectly sympatico with the syntactic focus of PET1. But, when learners have a choice of PETs, the advice should be tailored to the PET the learners are using. Learners using PET1 take a structural view and use syntactic terminology; thus making the advice relevant and understandable to the learner. Syntactic advice for PET2 users, however, would seem anomalous. PET2 users think in terms of input/output cases and probably need advice based on this alternative way of viewing programs (e.g. advice about trying to run their programs using input that will test a missing case). This points out an intriguing aspect of combining two different systems, such as PETAL and SCENT: each can take advantage of the capabilities of the

other to make both more effective in combination, as long as their mutual interdependencies are taken into account.

G.E.N.I.U.S.

Learners often solve their programming problems by discussing them with a knowledgeable expert, who is assumed to be providing relevant and insightful advice. Even if this advice is largely irrelvant and uninspired, as long as the illusion of experthood is maintained learners can still be helped. Much as the ELIZA system (Weizenbaum, 1966) could provide a convincing simulation of a psychotherapist under some circumstances, it should be possible to build a programming advisor that helps learners merely by encouraging learners to help themselves. The advisor need not be widely knowledgeable, but, as with ELIZA, only needs to keep learners cognitively engaged long enough to solve their own problems.

This is the insight underlying the design of the G.E.N.I.U.S. system (McCalla and Murtagh, 1991). G.E.N.I.U.S. is an advisor to help novice learners debug PL/C programs. G.E.N.I.U.S. interacts with a learner by producing explanations of error messages in the learner's code. Each error message number entered by the learner acts as an index into a node of a discrimination network. Associated with this node is either a natural language blurb explaining the error, or a question to ask the learner about the context of the error. Normally such questions are of the "yes/no" variety , the answer to which can be easily interpreted. Depending on the answer, G.E.N.I.U.S. branches to other nodes in the network where either further clarification is sought from the learner or an explanation is provided. With this simple architecture, and a moderate amount of fairly straightforward knowledge engineering to build the discrimination network, G.E.N.I.U.S. is able to produce quite realistic interactions, as can be seen in the sample input/output in Figure 7.

In an informal formative evaluation, G.E.N.I.U.S. was made available to a class of novice PL/C programmers. Most of the students in the class used the system only a few times before catching on that G.E.N.I.U.S. did not live up to its name. The problem was threefold: first, that G.E.N.I.U.S. often did not know about a particular error message entered by a learner due to limitations in stocking its knowledge base; second, that the system often was redundant; and third, that G.E.N.I.U.S. can be easily fooled by unexpected learner input. For an "ignorance-based" system like G.E.N.I.U.S. (or ELIZA) it is critical that the illusion of experthood be maintained. Once this is shattered by stupidity, redundancy or ignorance on behalf of the system, then the credibility, and hence the effectiveness, of the system is undermined. While it is relatively easy to solve the first problem by more thorough knowledge engineering, and the second through adding many alternative phraseologies for each blurb, the third problem is more difficult to overcome without facing the full range of AI problems confronting any normal "knowledge-based" intelligent tutoring system.

Thus, it is ambiguous from the G.E.N.I.U.S. experiment whether it is possible for an "ignorance-based" system to indefinitely maintain its social contract with the learner, especially when there are many co-temporaneous learners who can share knowledge of its faults among themselves. Nevertheless, in situations where learners are isolated (as in distance learning contexts) or in contexts where there are so many learners that even a relatively few successes add up to a considerable amount of help, systems like G.E.N.I.U.S. may be effective. It is also interesting to speculate on how many other domains there are where it is possible to take advantage of idiosyncracies in the social contract between the system and the user in order to provide a robust, easy to build, and at least somewhat effective ignorance-based learning environment.

G: You can ask me about:
syntax errors - the ones that produce error messages on your program listing
logic errors - no errors, but your program doesn't work
So what can I help you with?

S: syntax errors
G: Okay now, if I could get you to enter the error number of your first error message, we can begin solving your problem. Please include both the 2 letter prefix as well as the 2 digits. eg. sy02 is a sample error number.
S: cg1e
G: The variable that caused this error is an array or a string, right?

S: sure is
G: Hmm. Are you using it as a parameter in a function or procedure?

S: no
G: That's your problem then. Only parameters can use the star (*) in declaring their size (length of string or size of array). You'll have to declare that variable using an explicit number.
Are there any more questions that you have?

S: y
G: I'll need the error number of your next error message.
Please enter it below for me.

S: exe3
G: Okay, I have bad news for you. This error is usually caused by and infinite loop that is printing something over and over again. The problem lies in that the loop has no way of terminating like it should. Check your stopping conditions. I think that should pinpoint the problem. Okay?

S: okay
G: Does your mother know you think "okay".
But seriously, please answer my original question in a way I can understand.

S: y
G: Great! Glad I could help you.
Are there any more questions that you have?

Figure 7. - Sample G.E.N.I.U.S. Output

Conclusion

The single most important lesson of our research into "learning about programming" is the importance of making many bets. That is, it is very useful to try various approaches in order to attack the difficult problem of how to adequately support learners as they try to learn to program. By considering three different approaches we have been able to view the programming activity from three different perspectives: from the viewpoint of programming strategies (in SCENT), from the viewpoint of deep problem solving (in PETAL), and from the viewpoint of the surrounding social context (in G.E.N.I.U.S.). Given the complexity of the programming process, such a multi-layered view is critical to eventually building adequate support tools for programming. Combining ideas from these various perspectives has the potential to synergistically achieve even

greater effect, as our distance learning experiment shows. In fact, all three systems provide tools and ideas that can be applied to each other and to other domains. Case-based reasoning adds situation-specific knowledge to the SCENT advisor. The SCENT/PETAL symbiosis would seem to combine the best of both the scaffolding and deep diagnosis approaches. Incorporating more knowledge into G.E.N.I.U.S. could enhance the air of authority so crucial to its widespread success.

Another lesson of our work is to carry the research far enough that a system can be built that is complete enough to be empirically tested with real human learners. Each of the three systems described here, as well as the symbiotic distance learning system, has been tested with human subjects, albeit not extensively in some cases. It is only through use with real learners that theoretical ideas can be proven to be practical and that an educational system can be demonstrated to be more than a laboratory curiosity. In some cases this means years of work, as in the SCENT approach; in other cases, the ideas are relatively amenable to immediate practical application with human learners, as in G.E.N.I.U.S.

A corollary of this is to use the results of previous experiments in the design of the next system. This was done in PETAL, where successive systems were designed based on empirical work done using the previous system. This still has to be done with SCENT and G.E.N.I.U.S., which have only been empirically tested for feasibility, not for detailed lessons about how well they work, and where they fail to work. Ultimately, if systems with real world impact are to be built, theorizing, system building, and empirical testing must be mutually supporting enterprises.

In our future work we intend to continue exploring SCENT and PETAL, particularly in the distance learning framework. We have a specific need to take notice of our own lessons and do further empirical work. We are also convinced that the SCENT approach can be extended to all sorts of problem solving domains, and in fact have done preliminary work on providing advice for Unix shell system scripting and for a small part of a word processing domain, and to provide a help tool for a software engineering design environment. Thus, the AROMA knowledge engineering tools should be widely useful, even beyond the programming domain.

To date, we have not followed up on the G.E.N.I.U.S. approach. G.E.N.I.U.S.'s ability to take advantage of the environmental factors surrounding a learning experience is sympathetic with current interest in situated learning. In this sense, G.E.N.I.U.S. could be worthy of further work. Having been focussed on the knowledge needed to program, it would be interesting not only to see if ignorance-based learning could be made to work, but also if other contextually-based factors could be employed to improve systems designed to encourage learning about programming.

Acknowledgements

We would like to acknowledge the work especially of Shawkat Bhuiyan, who initially developed PETAL as part of his Ph.D. thesis; Randy Coulman, who has slaved for over two years to make SCENT robust, to build the AROMA development environment and, and who pioneered the use of case-based reasoning in SCENT; Andrew Dlugan, who souped up Bhuiyan's prototype PETAL system to be prettier and more functional; Gina Koehn, who helped to run the distance learning experiment; Peter Holt, who stimulated our interest in distance learning in the first place; and Kevin Murtagh, who designed and implemented most of the detailed G.E.N.I.U.S. protocols. The many others who worked on SCENT and numerous other projects under the auspices of ARIES over the years also deserve credit for contributing to the ideas underlying our research on programming environments for learning. Finally, we acknowledge the Canadian Networks of Centres of Excellence IRIS Project and the Natural Sciences and Engineering Research Council of Canada for financially supporting this work.

References

Bhuiyan, S.H., Greer, J.E. & McCalla, G.I. (1991). Characterizing, Rationalizing, and Reifying Mental Models of Recursion. *Proceedings of the 13th Meeting of the Cognitive Science Society*, Chicago, IL.

Bhuiyan, S.H., Greer, J.E. & McCalla, G.I. (1989). Mental Models of Recursion and Their Use in the SCENT Programming Advisor. *Proceedings of the Knowledge-Based Computer Systems Conference*, Bombay, India, 135-144.

Bhuiyan, S.H. (1992). *Supporting Students in the Use of Mental Methods of Recursion.* Ph.D. Thesis, Dept. of Computational Science, U. of Saskatchewan, Saskatoon, SK, Canada. (Also ARIES Laboratory Technical Report 93-4).

Coulman R.A. (1991). *Combining Case-Based Reasoning and Granularity for Educational Diagnosis in an Intelligent Tutoring System.* M.Sc Thesis, Department of Computational Science, University of Saskatchewan, Saskatoon.

Gentner, D. and Stevens, A., (eds.). (1983). *Mental Models.* Hillside, NJ: Lawrence Erlbaum.

Greer, J.E. & McCalla, G.I. (eds.). (1994). *Student Modelling: The Key to Individualized Knowledge-Based Instruction.* Berlin: Springer-Verlag.

Hobbs, J. (1985). Granularity. *Proceedings of the International Joint Conference on Artificial Intelligence*, Los Angeles, CA, 432-435.

Johnson, L. & Soloway, E. (1984). Intention-based Diagnosis of Programming Errors. *Proceedings of AAAI 84*, Austin, TX, 162-168.

Kahney, H. (1982). *An In-Depth Study of the Cognitive Behaviour of Novice Programmers.* TR 5, Open University, Milton-Keynes, UK.

Kurland, D.M. & Pea, R.D. (1985). Children's Mental Models of Recursive LOGO Programming. *Journal of Educational Computing Research*, 1, 235-243.

McCalla, G.I., Greer, J.E., & the SCENT Research Team. (1988). Intelligent Advising in Problem Solving Domains: The SCENT-3 Architecture. *Proceedings of the First International Conference on Intelligent Tutoring Systems (ITS '88)*, Montreal, Canada, 124-131.

McCalla, G. I. & Murtagh, K. (1990/91). G.E.N.I.U.S.: An experiment in ignorance-based automated program advising. *AISB Newsletter 75, United Kingdom*, Winter, 13-20.

McCalla, G.I.,Greer, J.E., Barrie, B. & Pospisil, P. (1992). Granularity Hierarchies. *International Journal of Computers and Mathematics with Applications* (Special Issue on Semantic Networks), 23, 363-376.

McCalla, G.I. (1992). The Central Importance of Student Modelling to Intelligent Tutoring. In E. Costa (ed.), *New Directions for Intelligent Tutoring Systems.* Berlin-Heidelberg-New York: Springer-Verlag.

McCalla, G,I. & Greer, J.E. (1993). Two and One-Half Approaches to Helping Novices Learn Recursion. In E. Lemut, G. Dettori, and B. du Boulay (eds.), *Cognitive Models and Intelligent Environments for Learning Programming.* Berlin-Heidelberg-New York: Springer-Verlag.

Price, R., McCalla, G.I. & Greer, J.E. (to appear). Combining Scaffolding and Diagnosis in a Distributed Tutoring System. *East-West Conference on Computer Technologies in Education*, Crimea, Ukraine.

Self, J. (1994). Formal Approaches to Student Modelling. In J. Greer and G. McCalla (eds.), *Student Modelling: The Key to Individualized Knowledge-Based Instruction.* Berlin: Springer-Verlag. 295-353.

Smith, R. (1986). The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations. *Proc. IEEE Workshop on Visual Languages*, 99-106.

Smith, R. (1993). A Prototype Futuristic Technology for Distance Education. In E. Scanlon and T. O'Shea (eds.), *New Dirrections in Educational Technology.* Berlin-Heidelberg-New York: Springer-Verlag. 131-138.

Soloway, E., Guzdial, M., Brade, K., Hohmann, L., Tabak, I., Weingard, P., & Blumenfeld, P. Technical Support for the Learning and Doing of Design. In M. Jones & P. Winne (eds.), *Adaptive Learning Environments: Foundations and Frontiers*. Berlin-Heidelberg-New York: Springer-Verlag. 173-200.

Weizenbaum, J. (1966). The ELIZA natural language understanding system, *CACM*, 9(1), 36-45.

White, B. & Fredericksen, J. (1985). QUEST: Qualitative Understanding of Electrical Trouble Shooting. *ACM SIGART Newsletter*, 93, 34-37.

Wills, L.M. (1990). Automated Program Recognition: A Feasibility Demonstration. *Artificial Intelligence*, 45, 113-171.