CREATING PLEASANT PROGRAMMING ENVIRONMENTS FOR COGNITIVE SCIENCE STUDENTS

M. Eisenstadt, J.H. Laubsch, & J.H. Kahney
Open University, England

1. Introduction and background

This paper describes our current efforts towards the systematic improvement of a LOGO-like software environment called SOLO (Eisenstadt, 1978), which has been used by over 1500 Cognitive Psychology students at the Open University. SOLO is geared towards the manipulation of assertional data bases, and provides the students with a handful of easy to use primitives with which they can address some elementary problems of knowledge representation. Students login to one of our DECsystem-20s from a regional study centre, and experience about 10 hours of hands-on activity early in the academic year. Later on, they attend a residential summer school at which they can acquire an additional 30 hours of hands-on experience.

Although SOLO is a toy language in some respects, the total user environment has many features which make it ideal for providing the vast majority of our students (80% of whom are computer-naive) with their first exposure to computing. Among these features are a spelling corrector, syntax-directed editing aids, automataic display of data base changes as they occur, "undo" facilities, and an easily modifiable user-profile.

An in-depth analysis of our students' errors (Lewis, 1980) has led to an improved design to help ensure that beginners can write syntactically correct programs with a minimum of fuss. A microcomputer implementation, which uses screen-oriented syntax-directed editing (cf. Teitelbaum & Reps, 1980) is being piloted just prior to this conference.

Even with SOLO's extensive user aids and carefully pre-tested curriculum materials, our students still experience problems in writing programs which perform precisely as intended. Because of this, we have undertaken a detailed analysis of their programming behaviour. Our empirical studies, described in section 2, have highlighted the use of a small number of programming schemas by a large proportion of our students. These schemas serve as the basis for an automated debugging assistant, which is described in section 3.

2. The behaviour of SOLO programmers

As part of their SOLO activities at the beginning of the year, our students are asked to "write a program which makes the following inference: If someone is found to be guilty, then whoever that person works for is also guilty." In solving the problem, students are invited to invent their own data structures and algorithms. We analyzed a sample of 160 student programs to see if some underlying order could be found among a potentially large variety of databases and program structures. As it turns out, the programs written by these students are built from a handful of basic program schemas. These schemas are langauge-independent programming constructs such as FILTER, CONJUNCTION, SIDE-EFFECT, and GENERATE-NEXT-OBJECT, which are closely related to those found in the LISP "plan library" of Shrobe, Waters & Sussman (1979).

The students' databases can be broken down into basic relational patterns. These patterns are reliably associated with particular program structures, allowing us to predict in 80% of the cases precisely what the students' program organization will be. To illustrate this point, consider the following typical (student-generated) database:

```
BURGESS                    PHILBY
 |---ISA--->COMMUNIST       |---ISA--->COMMUNIST
 |---WASAT--->CAMBRIDGE     |---WASAT--->CAMBRIDGE
 |---WORKSFOR--->PHILBY     |---WORKSFOR--->BLUNT

BLUNT
 |---ISA--->COMMUNIST
 |---WASAT--->CAMBRIDGE
 |---WORKSFOR--->THEQUEEN
```

This database exhibits the following patterns:

```
  Transitive-Relation:
        WORKSFOR (BURGESS PHILBY BLUNT THEQUEEN)

  Shared-Successor:
        (ISA COMMUNIST)    (BURGESS PHILBY BLUNT)
        (WASAT CAMBRIDGE)  (BURGESS PHILBY BLUNT)
```

Several items (BURGESS, PHILBY, BLUNT) are present in both the Transitive-Relation and the Shared-Successor lists. One of the program structures typically accompanying such a database structure contains three segments: a CONJUNCTION (COMPLEX-FILTER), SIDE-EFFECT, and GENERATE-NEXT-OBJECT. The Shared-Successors will be used as a FILTER selectively to choose nodes on which a SIDE-EFFECT is perpetrated (e.g. asserting X ISA SPY) and the Transitive-Relation list will be used to GENERATE-NEXT-OBJECT. Since the Shared-Successor list contains two patterns, a COMPLEX-FILTER will almost surely be used: IF x isa communist AND x wasat cambridge THEN ...

But why should a student write a program like this in the first place? Our analysis indicates that students have their own stylised interpretations, or mental models, of the task at hand. For instance, some students think that a program involving two inferences is called for: "If X has done something criminal then he is guilty. And if this is so, then his employer is subject to the same scrutiny, and so on for all employers." Other students feel that only one inference is called for: "Assuming X is guilty, his employer, by association, is guilty also, and so on for all employers."

The observed program structures ought to correspond to students' mental models of the task. Some of these mental models are "appropriate", in that they address the problem as stated, while others introduce certain anomalies which preclude a satisfactory solution. Such "inappropriate" models could actually be artifacts of students "thinking in SOLO" and getting led astray.

In order to test these ideas we have begun studying individual students in depth, collecting videotaped protocols and terminal session transcripts. The first subject began her project session with the clear intention of writing a program involving two inferences. Because of preconceived and inaccurate notions about constraints on the way she was allowed to approach the problem and because of misconceptions arising from her interaction with SOLO, she twice altered her intentions. At the end of the session the

student had a working program for a "one-inference" interpretation of the task described above, a compromise with which she herself was not completely satisfied. All of her programming behavior throughout this session, her various approaches to solving the problem, and the bugs she encountered, fell within the scope of the structures we had identified in the earlier analysis of 160 programs.

Our ability to categorize standard database structures and predict implementation strategies on the basis of those structures means that we can develop tools for assisting students in terms of the way in which they prefer to think about the task at hand. One such tool is described in the next section.

3. Debugging Aids

We have designed and partially implemented a tutorial debugging assistant which attempts to articulate the causes of bugs in terms which are close to the way we believe the students actually think about their own implementations. The bugs dealt with range from domain independent violations of the semantics of SOLO to domain specific errors that can be detected only if knowledge about the task at hand is used.

The debugging assistant uses symbolic evaluation (cf. Ruth, 1976) as a tool for (1) recognizing procedures as parts of a given "library plan", (2) detecting errors of the following types: unreachable program steps, purposeless steps, reference to absent database objects, infinite recursion because of a missing or unsatisfiable termination condition.

In the tutorial situation, a student's goal is to write a program to accomplish some modelling task. The debugging assistant is provided with a prototypical solution in terms of a canonical effect description. The task of the assistant is to recognize a match between the canonical effect description and an effect description derived from the student's own program.

In general these will not match, and the nature of the deviation will enable the assistant to draw the student's attention to shortcomings of his or her program which may be classified in the following way:

- The program will achieve the desired effect only in certain cases. A counter-example outside this range can provoke the student to discover the cause.

- The program would work if missing data or inconsistent entries in the data base were corrected. These corrections can be pointed out directly.

  A particular sub-procedure, if corrected using heuristics about typical errors (e.g. missing indirect link, violation of a program schema), would make the overall program correct. In this case, an appropriate hint can be provided for the student.

  None of the above.

In the last case, the student may initiate a dialogue, requesting help on a particular procedure. During the dialogue the assistant tries to find out the intended effect of that procedure (Eisenstadt & Laubsch, 1980). It does this by categorizing the procedure into one of several programming schemas stored in a language-independent "plan library", using a notation developed by Rich & Shrobe (1978).

Consider the case in which the nearest matching schema is "conjunctive filter and side effect". The assistant examines the deviation between the user's procedure and the stored schema. The following violations of the use of that schema may be recognized: omission of a conjunct, omission of the side effect, wrong (or transposed) arguments in the slots of the schema, or wrong control flow links. The assistant can describe these violations in terms which the student can relate to, since the library plans are themselves based upon schemas known to occur in students' code.

Since the students' procedures depend on their databases, and vice versa, the debugging assistant relies heavily on domain-specific knowledge to deal with alternative ways of formulating a solution to a given problem. Although the students have a great deal of freedom to choose ways of implementing solutions, they typically resort to a few common approaches. The assistant knows about these, and uses these both to make sense of what the students are attempting and to explain why they have gone astray.

4. Conclusions

Our experience with SOLO leads us to believe that a SOLO-like language/environment/curriculum could be of use to a broader group of cognitive science students-- for instance as the basis of a beginners' LISP curriculum oriented entirely towards pattern-matching and assertional data bases. For this to become a reality, it is important to understand precisely what remaining problems students have in this type of environment, and why. Our empirical work is a step in this direction. It has immediate spinoffs in that it provides a foundation for our debugging assistant. The assistant provides students with a tool for attaining their goals, and provides us with a tool for analyzing and describing their behaviour.

REFERENCES

Eisenstadt, M. Artificial intelligence project. Units 3/4 of Cognitive psychology: a third level course. Milton Keynes: Open University Press, 1978.

Eisenstadt, M. & Laubsch, J. Towards an automated debugging assistant for novice programmers. Proceedings of the AISB-80 conference on Artificial Intelligence, Amsterdam, 1980.

Lewis, M. Improving SOLO's user-interface: an empirical study of user behaviour and proposals for cost-effective enhancements to SOLO. Technical report no. 7, Computer Assisted Learning Research Group, The Open University, 1980.

Rich, C. & Shrobe, H. Initial report on a LISP programmer's apprentice. IEEE Transactions on Software Engineering, SE-4:6, 1978.

Ruth, G.R. Intelligent program analysis. Artificial intelligence, 7, 1976.

Shrobe, H., Waters, R., & Sussman, G. A hypothetical monologue illustrating the knowledge underlying program analysis. MIT Artificial Intelligence Laboratory Memo 507, 1979.

Teitelbaum, T., & Reps, T. The Cornell program synthesizer: a syntax-directed programming environment. Technical report 80-421, Department of Computer Science, Cornell University, 1980.