TEACHING PROLOG USING INTELLIGENT COMPUTER-ASSISTED INSTRUCTION AND A GRAPHICAL TRACE

by

EARL FOGEL

B.Sc. University of Saskatchewan, 1980

Bachelor of Journalism, Carleton University, 1982

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

February, 1988

3 9

Abstract

Two methods for improving the quality of Computer Assisted Instruction are examined. They are: using Intelligent Computer Assisted Instruction techniques to make the CAI system more flexible, and using graphics to increase the efficacy of teaching.

Two computer systems for teaching the Logic Programming language Prolog were developed.

The first is an ICAI system which uses the prerequisite relationships of the course material to plan a course of study. It distinguishes between methods of instruction and topics of instruction, giving students a great deal of freedom in choosing either one.

The second is an animated trace which graphically illustrates the execution of Prolog programs. Information is displayed in three windows -- one for Prolog goals, one for the database, and one for output from the program being traced.

Results indicate that ICAI and graphics can both be used effectively in the teaching of programming languages, particularly in combination.

TABLE OF CONTENTS

List of Figures

Acknowledgements

The author would like to thank everyone who tried out the two systems and gave advice during their development.

Many of the ideas for teaching Prolog came from the excellent Prolog courses given by Ray Reiter and Harvey Abramson at UBC, from the Programming in Prolog text by Clocksin and Mellish, and from the CProlog Users Manual by Fernando Pereira et al.

Alan Mackworth, who supervised this research, contributed leads to much of the ICAI literature and, with great perseverance, convinced me of the advantages of graphics in teaching programming languages.

Richard Rosenberg has offered some valuable advice for the final draft of this thesis, and his assistance is sincerely appreciated.

Financial support for this research was provided by the Natural Sciences and Engineering Research Council of Canada.

Chapter One

Introduction

For more than 20 years, people have been writing Computer Aided Instruction (CAI) programs. Most of them are not very good. Hofstetter (1981) estimates that as many as 5000 out of the 7000 hours of instructional software written for Plato are useless, yet Plato is described as one of the most successful CAI systems.

This chapter will show why computers are such poor teachers, and will put forward some ideas about what can be done to improve them. Two computer systems that illustrate these ideas will then be described.

1.1 Why Computers are Such Terrible Teachers

Consider the following locked room analogy.

> Imagine you are sitting all alone in a locked, soundproof room, with nothing but a typewriter, some paper, and a book. On one wall of the room are two slots, labelled In and Out.
>
> Your job is to teach the contents of that book to anyone who happens to be on the other side of the wall, by passing notes through these slots.
>
> You can know nothing about the subject of the book, you must give up most of your command of the language in which it is written (and which the students use), and you must forget everything you know about teaching. (After Searle 1983).

Under the circumstances, it would be difficult for anyone to teach effectively, yet these are precisely the conditions under which most CAI programs operate.

Knowing nothing about the subject they teach, they can only present the student with previously written pages of text. Knowing nothing about the student, they cannot tailor their presentation to his or her needs. Knowing nothing about teaching, they are unable to vary their teaching strategies to best suit a particular topic or a particular student. Further, the only contact most CAI programs have with the real world is through the terminal screen and keyboard. Students cannot pick up visual or voice-tone cues from the program, nor can it pick up such cues from them.

In summary, a typical CAI program does not understand what it is teaching, who it is teaching, or how to teach. With these handicaps, it is no wonder computers are such poor teachers. In fact, it is a wonder they are as good as they are.

1.2 Why Computers are Such Good Teachers

There are good CAI programs. There are, for example, about 2000 hours of Plato courseware (educational software) that Hofstetter did not dismiss as useless.

Some CAI programs succeed because their subjects are especially well suited to computer instruction. A few years ago, one study found that "95% are arithmetic programs" (Ragsdale 1982).

Much of CAI focuses on arithmetic because arithmetic is easy to teach with computers. It is easy because computers can do arithmetic, and students can watch them doing it.

Indeed, a computer can help a student with arithmetic, and can compare the student's answers with its own.

Another promising subject for CAI is computer programming. In this area too, the computer can check the student's answers against its own. Looi has written a program which teaches the Pascal assignment statement (Looi 1984). His system checks the syntax of simple programs written by the student, runs them on predetermined test cases, and verifies the results.

Other CAI programs succeed because they use simulations. With these, the student learns about a real-world system by experimenting with a model of that system. The models include economic systems, with the student playing the stock market or running a Third World country; or biological systems, showing population changes in fish or bacteria.

These programs all have one thing in common -- competence in their domain of expertise. They are unable to reason about their domain, nor can they explain it, but there is enough domain knowledge built into these programs that they can perform competently within it. Going back to the locked room analogy, these programs are successful because they know something about the subjects they are teaching.

1.3 Making Computers into Better Teachers

Efforts to make computers better teachers can be classified into two categories:

* Allowing a broader spectrum of interactions with the outside world (unlocking the room).

* Improving their intelligence and their knowledge of the world (putting someone in the room who knows the student, who knows the subject being taught, and who knows how to teach).

1.3.1 Unlocking the Room

This involves breaking down the barriers both between the teacher and student, and between the teacher and the subject being taught.

Human teachers can talk, gesture, draw pictures, show movies and use computers to put ideas across, while students can talk, groan, scratch their heads, and so on. In the most basic form of CAI, on the other hand, the only means of communication is text, typed at the keyboard and displayed on the screen. The use of light pens, touch sensitive screens, slide projectors, videotapes, speech synthesis and graphical displays can all enrich this interaction.

One can also enrich the interaction between the computer and the subject it teaches. A human teacher often has personal experience with the subject being taught, and can support the lessons with real world demonstrations. He or she can turn over a leaf, or open up the hood of a car. It is easier to teach about trees, for example, if you know something about trees, and if you can point to a tree while you talk. Computers are good arithmetic teachers because they can show the student how to do arithmetic, and can check the student's work against their own.

One way to teach about a domain with which you can't interact is given by Alan Bundy's notion of "stories". A story is a model or an analogy which makes it easy to grasp the central ideas behind the subject being taught. Learning is much easier with a good story.

Here, Bundy writes about stories for teaching programming languages:

> [It] is important to give a model of what the computer will do with his/her programs. The student must be able to anticipate the effect of running his/her program, otherwise he/she will be unable to design it, debug it, modify it, etc. ...
>
> When teaching LOGO to school children, Tim O'Shea and Ben du Boulay found the provision of a suitable model to be central to the design of the course and the language interface (Bundy, 1983).

In teaching about turtle graphics, one has the analogy of live turtles crawling around on the floor, and one has a computer which simulates the turtle on a screen.

To summarize, one can make computers into better teachers by improving student-computer communications, by improving domain-computer interactions, or by using a good story.

1.3.2 Putting a More Intelligent Teacher in the Room

Most CAI programs lack the knowledge to teach effectively. They don't understand students and they don't understand what they are teaching. Making computers into better teachers, by giving them knowledge of the real world, and telling them how to use it, comes under the domain of Artificial Intelligence.

Researchers who try to put knowledge about students, about teaching, and about what is being taught into CAI programs call their field Intelligent Computer Assisted Instruction (ICAI); presumably to distinguish it from the not so intelligent kind of CAI.

There are four general areas of ICAI research:

* Increasing the computer's knowledge of students (with a student model).
* Increasing its knowledge of the domain (with a domain model).
* Increasing its knowledge of teaching and learning (with an educational model).
* Increasing its ability to carry out a natural language dialog with students.

Each of these areas involves many interesting subproblems. Within student modelling, for example, there is the problem of finding out what students already know, what they don't know, and what misconceptions they have, as well as the problem of changing the model as the student learns (Self, 1974).

1.4 Teaching Programming Languages

One good domain for CAI is computers themselves. Computers can't do much with trees and cars, but they are ready made to teach about computers. A computer can't point, but when it teaches about tape drives, it can spin tapes back and forth. When it teaches computer programming, it can trace programs, and it can check and display the results.

Teaching programming languages is made especially easy because people have invented many stories for programming: things like flow charts, algorithms, data flow diagrams, modular design, and interactive trace programs.

1.5 Teaching Prolog

Prolog is a Logic Programming language. The execution procedure for a Prolog program is largely embedded in the Prolog interpreter, and not in the program itself. Whereas programs in traditional programming languages have an explicit control structure (with sequential execution, loops, and so on), Prolog programs are mainly descriptive, based on the implicit proof procedure of the interpreter (top down depth-first search with backtracking).

Because it differs from traditional programming languages, Prolog requires different stories. Some Prolog stories are evaluated by Bundy (1983). These include: Or Trees and And/Or Trees (both from Kowalski, 1979); the Byrd Box, Arrows, the Flow of Satisfaction, and a Prolog trace program (all described in Clocksin and Mellish, 1981).

All the benefits of teaching programming languages hold for Prolog. Students can write programs, which the computer can run and trace, and there are good stories for teaching Prolog. As well, Prolog's declarative semantics make it a very nice vehicle for the use of ICAI techniques. In Prolog, it is easy to define rule-bases for educational, student, and domain models, and to use these in decision making.

1.6 This Thesis

Two computer systems for teaching Prolog were developed for this thesis.

The first is an ICAI program that teaches Prolog. It uses knowledge of the student, the domain, and of teaching to determine which topic to teach, and how to teach it. The second provides an animated trace of the execution of Prolog programs. Students can use it to follow the execution of their own programs, and of the programming examples provided by the ICAI program.

Both programs were developed on a DEC VAX 11/780 running Berkeley UNIX. The ICAI program is written in CProlog. The trace is written in CProlog and in C, using the CURSES windowing package, which in turn uses the UNIX Termcap terminal database.

1.7 The Prolog ICAI System

1.7.1 And/Or Prerequisite Trees

The use of And/Or Trees to represent prerequisite relations for a CAI course is described in a number of papers by Darwin Peachey and Gordon McCalla at the University of Saskatchewan (Peachey, 1982) (McCalla et al, 1982).

Briefly, the course to be taught is divided into an number of separate topics, linked together by prerequisite relationships. The And/Or Tree links the topics of the course.

Some topics have no prerequisites -- these may be taught immediately. The others have prerequisites which should be covered first.

A topic may have more than one prerequisite -- all of which are required. These are connected by AND links in the prerequisite tree. Alternatively, a topic may have several prerequisites -- only one of which is required. These are connected by OR links.

Here is a part of the prerequisite tree (actually a directed graph) from the Prolog CAI course:

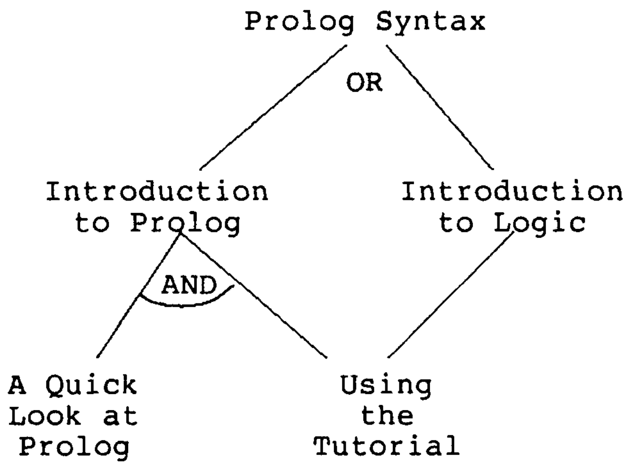


Figure 1: Example of And/Or Prerequisite Tree

In this example there are two ways to satisfy the prerequisite requirements for "Prolog Syntax". Either study "Introduction to Prolog" along with both of its prerequisites, or study "Introduction to Logic" and its single prerequisite.

1.7.2 Who Makes the Decisions?

Using AND/OR Trees lets the system make reasonable choices for topics to be studied, but care must be taken to ensure that the choices made are not too authoritarian. The system makes decisions based on incomplete information, and students must be able to overrule these decisions when they choose.

A major consideration in the current research was to produce a system that gave the student a great deal of freedom of choice as to what to study, how to study it, and when.

The Prolog CAI system chooses topics for the student to study, and chooses methods for teaching those topics. These choices act as defaults. Each student is free to accept the systems choices or to disregard them to pursue his or her own interests.

1.8 The Animated Trace (Anilog)

Anilog is a window-oriented trace for Prolog programs (the name comes from ANImation of LOGic). It displays information about the execution of Prolog goals, their success, failure, backtracking, and recursive calls to other goals, as well as showing the database clauses they use, and any output they generate.

All of this information is displayed on the screen, in three windows: one for goals, one for the Prolog database, and one for user output.

1.9 Sample Protocols

The following three pages show the beginning of a typical session with the Prolog ICAI system. Following this is a snapshot of the animated trace, part-way through satisfying the goal:

```
?- write(hi), member(elf,[dwarf,elf,pixie]), fail.
```

A more complete example of the animated trace may be found in Appendix 4.

```
     PROLOG COURSE - TABLE OF CONTENTS

     Using This Tutorial
     A Quick Look at Prolog
     Introduction to Prolog
     Introduction to Logic
     Syntax
     Semantics
     Unification
     Proof Procedure
     Side-effects
     Prolog Basics
     Built-in Predicates
     Introduction to the Builtins
     Arithmetic
     Input/Output
     I/O Basics
     File Access
     Character I/O
     Term I/O
     Reading-in Programs
     Convenience
     Operators
     Control of Execution
     The Cut
     Comparison of Terms
     Meta-Logical
     Debugging
     Sets
     Program Information
     Changing the Data Base
     Internal Data Base
     Environmental
     Definite Clause Grammars

Type <cr> and the system will choose a topic for you,
(or type h for help).

|:
               [The user types a carriage return,
                so the system chooses a topic]
```

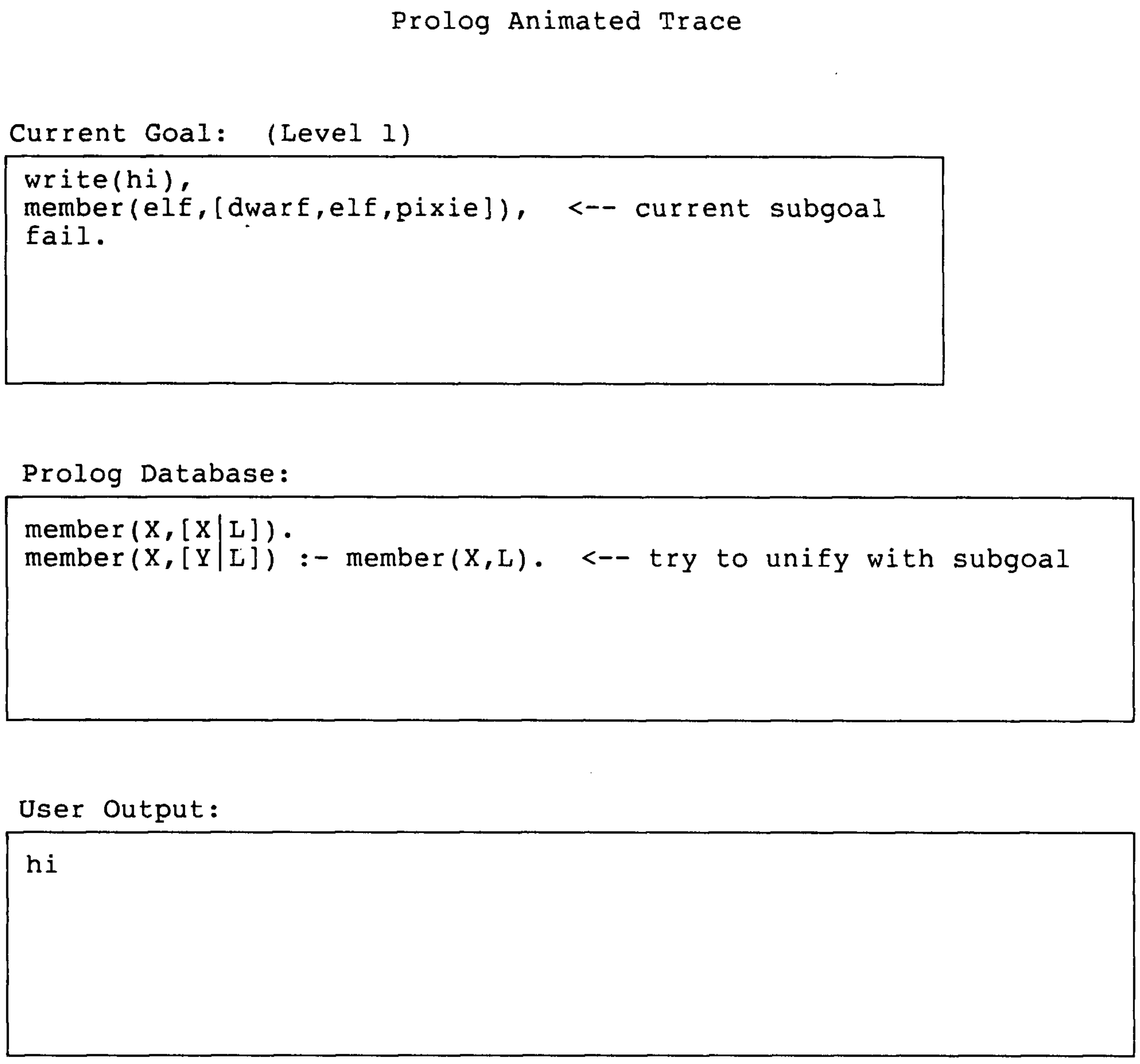
Figure 2: The start of the Prolog ICAI course

```
                        Prolog Tutorial System

      *** A Quick Look at Prolog ***

      Options:

           1 - lesson

           2 - example









Please choose one of the above options (or type h for help)

|:
                [The user types a carriage return,
                 so the system chooses an option]
```

Figure 3: The first topic selected

```
                   Welcome to CProlog


  Prolog attempts to answer questions based on the
information it has been given (its data base).

  A statement in Prolog is called a clause.  Here
is a small database, consisting of 5 clauses. The
comments to the right indicate the intended meaning
of each clause.


greek(souvlaki).              /* souvlaki is greek    */
greek(socrates).              /* Socrates is greek    */
human(socrates).              /* Socrates is human    */
human(descartes).             /* Descartes is human   */

philosopher(X) :- human(X).   /* all humans are
                                philosophers         */

   The symbol  :-  means "if", or "is implied by",
or "can be proven by", so the last clause above
can be read as:


   - X is a philosopher if X is human.
   - X is human implies X is a philosopher.
   - To prove that X is a philosopher, first
     prove that X is human.


   Prolog knows that socrates is the name of a particular
 object, while X can be any object because X begins with
 a capital letter.  In computer programming terms, X is a
 variable.

   A variable is a kind of place holder or blank space
 into which Prolog tries to put the names of objects.


   Type <cr> to continue (or h for help).

   |:
```

Figure 4: The first lesson selected



```
                     Prolog Animated Trace

Current Goal:  (Level 1)
 write(hi),
 member(elf,[dwarf,elf,pixie]),   <-- current subgoal
 fail.

Prolog Database:
 member(X,[X|L]).
 member(X,[Y|L]) :- member(X,L).  <-- try to unify with subgoal

User Output:
 hi
```

Figure 5: Snapshot of the Prolog Animated Trace

Chapter Two

Literature Survey

This chapter surveys various attempts to improve the use of computers in education. More details on these and other projects can be found in Kearsley (1987), Jones (1986), Yazdani (1984), Sleeman and Brown (1982), and Barr and Feigenbaum (1982).

Our discussion is divided into the two areas discussed in Chapter One:

1) Using more intelligence. We will look at four areas: understanding students, understanding the domain of instruction, understanding how to teach, and using natural language dialogue.

2) Broadening the interaction between teacher, student, and the domain of instruction. We will look at two areas: teaching in suitable domains, and using graphics to improve communication.

Of course, some systems fit into more than one category. Guidon (Clancey 1979) for example, is discussed under Domain Understanding, but it could equally well have been in the section on Understanding Students. Similarly, WUSOR (Goldstein 1979) could be discussed in three different places, because its Genetic Graph is student model, domain model, and educational theory all rolled up into one.

2.1 Using More Intelligent Teachers

2.1.1 Understanding Students

For teaching purposes, it is important to have a model of the student's knowledge of the subject being taught. A teacher (or ICAI system) compares what he or she believes the student knows with what he or she would like the student to know.

The simplest and most common type of student model is simply a record of which lessons have been studied. The next step up is a record of which topics have been learned.

Goldstein (1979) suggests a more complex student model which combines knowledge of the domain of study with knowledge of the learning process. He uses this "Genetic Graph" in a program called WUSOR, which is a tutor for the computer game WUMPUS.

The Genetic Graph models the evolution of a student's knowledge. Nodes in the graph represent the procedural skills that a student acquires in changing from a novice into an expert player of WUMPUS. Arcs in the graph represent the learning processes that are used to acquire these skills. These processes include: analogy, specialization, generalization, prerequisite and deviation.

A student's knowledge of the game at any given time is represented by a subset or perturbation of this graph. The system decides what to teach, and how to teach it, by following arcs from the area the student has mastered to new areas of the graph. The new nodes (or skills) are then added to the region representing the student's knowledge.

It is very difficult to find out what is going on in the minds of students. It is all very well to say that the student model represents what the student knows, but how can an ICAI system find out what a student does know?

"Why Your Students Write Those Crazy Programs" (Soloway et al. 1981) describes some of the mistakes made by students in a beginning Pascal class, and speculates on the reasons for them. Proust (Johnson and Soloway 1987) is a program which finds bugs in Pascal programs. Proust finds bugs by comparing the program with a formal description of the problem the program is intended to solve.

Matz (1982) lists some common reasons student errors in high school algebra problems. These include extrapolating old rules to fit new situations when the old rules do not, in fact, apply, making errors through carelessness, and not having enough knowledge to deal with the problem.

Burton and Brown (1977) did something a similar study for simple arithmetic. They listed over 100 common mistakes children make in performing two-column subtraction problems, and there is no reason to believe that all the mistakes have been found. Their systems, BUGGY and later DEBUGGY, are aimed at diagnosing the problems students have; they do not go so far as to plan strategies for correcting the student's misconceptions.

Colburn (1982) is also concerned with diagnosis, this time in the area of reading problems. She proposes an expert system to advise a teacher or counsellor in diagnosing reading problems in children. The system uses a database of diagnostic rules to recommend tests and to analyse their results. Like BUGGY and DEBUGGY, this system diagnoses problems, but it does not go so far as to prescribe corrective action.

2.1.2 Understanding the Domain of Instruction

A simulation is one type of domain model. It can be used by students to see how a real-world system reacts to different conditions. For example, it can demonstrate how an airplane reacts to having its wing flaps raised.

ThingLab (Borning 1979) is a general purpose simulation toolkit. It provides a language for defining simulations in terms of part-whole hierarchies and inheritance structures, in terms of the relationships between parts of the model, and in terms of constraints on those relationships.

ThingLab also provides a graphical display for simulations. The simulated system can be modified, and observed, by interactively changing parts of the display, and seeing what happens. For example, a Fahrenheit to Celsius converter might be displayed as two interconnected thermometers. Lowering the reading on the Fahrenheit thermometer causes a corresponding reduction on the Celsius scale, and vice versa.

A more direct way to use a domain model in teaching is to use it to understand what is being taught, and to use that understanding to provide explanations to students. These can be summaries of particular concepts, or descriptions of the relationships between concepts. To date, most systems that use domain knowledge do not model the entire domain, they model its structure. Going back to the airplane example, such a system would know how raising wing flaps affects air speed, but would not know what a wing flap was.

Stephens et al. (1982) use reasoning about the structure of the domain to teach about climate. Essentially, their domain model gives cause and effect relationships between such environmental factors as warm ocean currents, rainfall, mountains, wind direction, and warm and cold air masses. The system does not understand mountains, but it does know how they affect air currents.

SOPHIE (Brown et al, 1976) teaches students how to diagnose faults in electrical circuits. It contains a simulator for electronic circuits, which it uses to see how reasonable a student's troubleshooting strategies are. When it finds things in its model that the student does not seem to understand, it can point them out.

GUIDON (Clancey 1977) teaches diagnostic skills to medical students by leading them through selected case studies. It keeps track of expressed student interests, and uses them in choosing cases to present.

Guidon's domain model is interesting because it is an entirely independent expert system. MYCIN is an expert system that diagnoses certain kinds of blood disorders, using a database of diagnostic rules to make its decisions.

Guidon teaches about MYCIN's rules. It runs MYCIN to obtain a diagnosis for a particular case. Then Guidon goes through the same case with a student. It compares the student's diagnostic procedures with those of MYCIN, and it explains MYCIN's diagnosis to the student. To do this, it uses a database of explanations of MYCIN's rules, and another database of teaching rules.

Guidon2 (teaching about NEOMYCIN) adds yet another database. This contains rules for analyzing the student's behavior (Clancey 1979) (Clancey 1987). It attempts to uncover the diagnostic process the student is using, so that mistakes in that process can be pointed out and corrected.

Kimball's symbolic integration tutor (Kimball 1973) uses structural knowledge in a very different way. It guides the student through the solution of symbolic integration problems. When a solution is produced, it compares it to a previously stored solution. If the new solution is shorter than the old one, it is deemed to be better, and the system adopts it as the new standard.

Suppes (1972) suggests the use of "strands" to structure the domain of instruction. In the school system, students often study a subject at different levels in different grades, and a strand corresponds to such a subject.

Suppes' strands can be though of as towers, with simple problems at the bottom, and more complex ones higher up. A student can be doing problems at one level on the arithmetic strand, and be at quite a different level on the others.

McCalla's Lisp Course (McCalla et al. 1982) is a general purpose system using the prerequisite relationships between concepts to guide its teaching. Concepts in the Lisp Course are linked together by an And/Or Prerequisite Tree. So, for example, the basic concept of recursion is a prerequisite for both tail recursion and for indirect recursion, and these in turn are prerequisites for a mastery of the entire recursion topic. The Lisp Course is described in more detail in Chapters One and Three.

The Scent automated advisor (McCalla et al. 1986) is intended to aid students in debugging Lisp programs. Using knowledge of Lisp, knowledge of general-purpose programming techniques, and knowledge of the specific task at hand, Scent analyses student programs in a variety of ways.

Scent is organized into several components, which communicate through a "blackboard". Program behavior components produce traces and cross-reference listings; strategy judges attempt to determine which solution strategy is being used; diagnosticians look for errors in strategy; while task experts look at how well the program is solving the particular task at hand.

2.1.3 Understanding How to Teach

LOGO (Papert 1980) is based on discovery or Piagetian learning (after Jean Piaget, an educational theorist). Discovery learning is natural learning, without effort or teaching. An example of this is the way children learn their first language -- by hearing it and being interested, not by studying it.

The original LOGO was a simple computer language which was used to control a mechanical turtle. The turtle rolled around on the floor, and it had a pen in its belly, which could be raised or lowered. LOGO commands told the turtle to take so many "turtle steps" forward, or to turn, and moving with the pen down would draw a picture.

More recent versions of LOGO propel a graphical turtle around a computer terminal display. Children can play with the turtle, making it draw different pictures. They can also play in micro-worlds. In one such micro-world, a turtle in motion tends to remain in motion, while a turtle at rest tends to remain at rest.

Another extension to LOGO has been the inclusion of some of the basic list handling functions of LISP. LOGO is now a popular first programming language, and is available on many micro-computers.

Some of the people who created LOGO are now working on a new program called Boxer (DiSessa 1986). Designed to make the activity of programming more accessible to students, Boxer is based on one uniform metaphor -- the box. In Boxer, programs, data, environments and sprites are all represented visually as boxes. A program box inside another program box represents a subroutine, while a data box which contains other data boxes represents a record structure. One of the design criteria behind Boxer is the principle of "naive realism": the appearance of the system should accurately reflect its underlying structure, so that an understanding of the appearance of the system "can be translated directly into an understanding of the system". Boxes (and hence programs, data, etc.) can be altered by direct manipulation of their on-screen representations.

O'Shea's (1979) Quadratic Tutor learns as it teaches. It changes its own teaching rules in an attempt to select the most effective teaching strategy. It can, for example, be directed to optimize its strategy so as to decrease the time a student spends with the tutor.

2.1.4 Natural Language Dialogue

Dialogue, in which the student and program talk to each other in natural language, is both one of the earliest goals of ICAI and one of the furthest from achievement.

Carbonell (1970) wrote the first dialogue system, called SCHOLAR. SCHOLAR taught geography. Its knowledge of the subject was stored in a semantic network, which it used to generate questions for students, to check their answers, and to answer questions posed by students. Carbonell called this sort of interaction, which was sometimes guided by the program and sometimes by the student, a mixed-initiative dialogue.

A more recent attempt at dialogue has been made by Curran (1982). He, along with students in an Artificial Intelligence course, wrote a "teacher/learner". This program knows some things about the domain of computer science, and it wants to learn more. It has "a thirst for obtaining Computer Science information". Curran's program is not intended to teach computer science, but to teach Artificial Intelligence. Students study how the program works, not what it knows.

The program engages students in a simplified natural language dialogue, modelled after Weizenbaum's Eliza program (1965). It "makes the machine appear more clever than it is". The program "can be temperamental and change the subject, or respond with moody sentences reflecting any of several emotional states" (quotations from Curran, 1982).

The program gives more credibility to information that comes from several sources, and less to information when it is contradicted. Furthermore, individuals who frequently input believable information are deemed more trustworthy than those who often enter contradictory items. The program can construct general rules from specific information (unless and until it finds a counter example). Finally, it "forgets" information which is not very believable, or which is not frequently accessed by students.

2.2 Broadened Interactions among Teacher, Student and Domain

2.2.1 Teaching Programming Languages

The main benefit of teaching about a programming language is that the teaching program and the student can both run sample programs, providing a ready made domain model. On the other hand, it brings its own special problems as well, particularly understanding programs that students write.

The Basic Instructional Program, or BIP, (Dageforde et al. 1978) teaches programming in Basic through the use of author-supplied example problems. The student writes a program to solve a given problem, then BIP runs it and compares the results with a previously stored solution.

BIP stores information about the skills needed to solve each problem in a Curriculum Information Network. It chooses problems for a student by looking for ones that use one new skill, along with several skills the student already has.

Soloway and his colleagues (1983) have investigated program understanding in their system MENO-II. It analyses student programs, and tries to catch run time errors, both those that are problem dependent, and those that are problem independent. A special Problem Description Language (PDL) is used both by the student for program development, and by MENO for program understanding.

MENO compares the PDL description of the student's program with a stored description of a bug-free version of the same program, by matching corresponding program structures (eg. loops). It can only cope with a few control structures, specifically straight line code, branching, and simple loops -- the sort of things beginning programmers use.

Laubsch and Eisenstadt (1981) propose a similar approach to program understanding. Their system attempts to translate programs written by students into "plan diagram notation". This encodes control flow and data flow information. It detects "unreasonable code", such as unused variables and duplicate statements. Then it tries to match the description of the student's program with one from its library.

2.2.2 Graphics

Antics (Dionne and Mackworth 1978) was developed for a M.Sc. Thesis at the University of British Columbia. It is used to produce animated films showing the execution of LISP programs. Antics graphically traces the evaluation of LISP functions, taking information about the S-expression being traced, the flow of control, and the assignment of values to variables, and displaying it on different parts of the screen.

Antics uses graphics to make programs written in a non-graphical language (LISP) easier to understand. The natural next step is to abandon the original language and to use the graphical representation directly for programming.

This is what Lakin proposes (Lakin 1980). LISP is a symbol processing language, whose symbols are strings of text. Lakin's system Pam (for PAttern Manipulation) is a text-graphics processing language. Programs in Pam are a mixture of text and graphics, and the objects they process can likewise be a mixture of the two.

2.3 Summing up the Literature

The systems examined in this chapter are largely experimental in nature. "Because of the size and complexity of ICAI programs, most researchers tend to concentrate their efforts on the development of a single part of what would constitute a fully usable system" (Barr and Feigenbaum 1982). It is not surprising, therefore, to find that few have found their way out of the laboratory and into everyday use.

One aim of this thesis is to develop a practical ICAI system, and it is with this in mind that the following evaluation is made.

There have been few practical advances in student modelling. Soloway, Matz, Burton and Brown, and Colburn have each taken some steps towards the diagnosis of student misconceptions, but none of them has produced a complete system which can teach as well as diagnose.

Goldstein and McCalla, with less ambitious student models, have each produced working experimental ICAI programs.

It is interesting to compare Goldstein's Genetic Graph with McCalla's And/Or Prerequisite Tree. Both place the concepts to be learned into a directed graph. Both represent the student's knowledge with a subset of this graph, and both represent learning by following arcs from the known territory to the unknown.

The main difference between the two is that an arc in the Genetic Graph represents the learning process that a student is believed to use in traversing it, while an arc in the And/Or Tree simply represents a prerequisite relationship.

The Genetic Graph is a more ambitious approach, but it seems to make less sense for a practical system. By attempting to anticipate the student's learning processes it is overly restrictive (expecting a student to generalize in one case and to use analogy in another). The And/Or Tree leaves the learning process, and the method of instruction, more open and more flexible for individual students.

The domain models described in this chapter are of two types. The type used in SOPHIE to teach electronic troubleshooting uses a relatively deep knowledge of the domain to show students the results of their actions. The other kind, used in WUSOR and in the Lisp Course, simply model the structure of the domain to show how different topics are related.

In the long run, the greatest advances in ICAI may come from research into new educational theories or from the use of natural language dialogue. For the present, however, the impact of these areas on practical systems remains slight.

LOGO is the only system described in this chapter that has come into widespread use, and its success can perhaps be attributed to three factors. Rather than using inadequate student and domain models to predict what the student should be studying (often incorrectly), Logo's discovery learning technique lets the student decide. It has an *appropriate domain* (computer programming and problem solving), which is large enough to be worth discovering, yet tractable enough that students can do much of the exploring on their own. Finally, Logo uses *graphics* to show students what their programs are doing.

Chapter Three

Prolog ICAI System Design

This chapter describes the design of the Prolog Computer Assisted Instruction program. The Animated Trace program will be described in Chapter Four.

3.1 Design Goals

The Prolog ICAI system was intended as a practical tool for learning Prolog. As such, it is more important for it to be easy to use and complete, than to be innovative. When concepts from ICAI could make the system more flexible and useful, they have been incorporated into the design. When it seemed they would detract from the system's effectiveness or its ease of use, such ideas were not incorporated.

The system was also designed to be non-authoritarian. While an attempt was made to have the system make intelligent decisions, students often have a better idea of their own needs than the system does, so it is important to let students overrule the system when they want to.

The Prolog ICAI system's design is independent of the subject matter of the course it is teaching, and is also independent of the instructional methods used to teach any individual topic.

Finally, it was hoped that the system would be interesting. That is, students should enjoy using it.

3.2 General Design

At the highest level, the system's design is very simple. First it chooses a topic to teach. then it chooses a way to teach it, and then it teaches it. This cycle repeats until the course has been completed.

To make these choices, the system consults a list of the course topics, a prerequisite structure (described below), and a list of the instructional methods available for each topic.

In order to use the system to teach some other course, one need only change the topic list, the prerequisite structure, and the instructional modules themselves.

Throughout the course, the <return> key is used to let the system make decisions. By continually pressing <return>, a student can progress through the entire course, with the system choosing all the topics to be studied, and the methods for studying them. On the other hand, a student who wants to guide his or her progress is free to do so. The system's choices of topic and method are only defaults. Students can always:

- Choose a topic or a method of instruction for themselves.
- Go into CProlog to try out something they have learned.
- Suspend the Prolog course, and resume it later on, exactly where they left off.
- Review a topic, or review the prerequisites for a topic.
- Try alternate methods of instruction, or alternate prerequisites for a topic.

The following diagram illustrates the design of the Prolog CAI system.

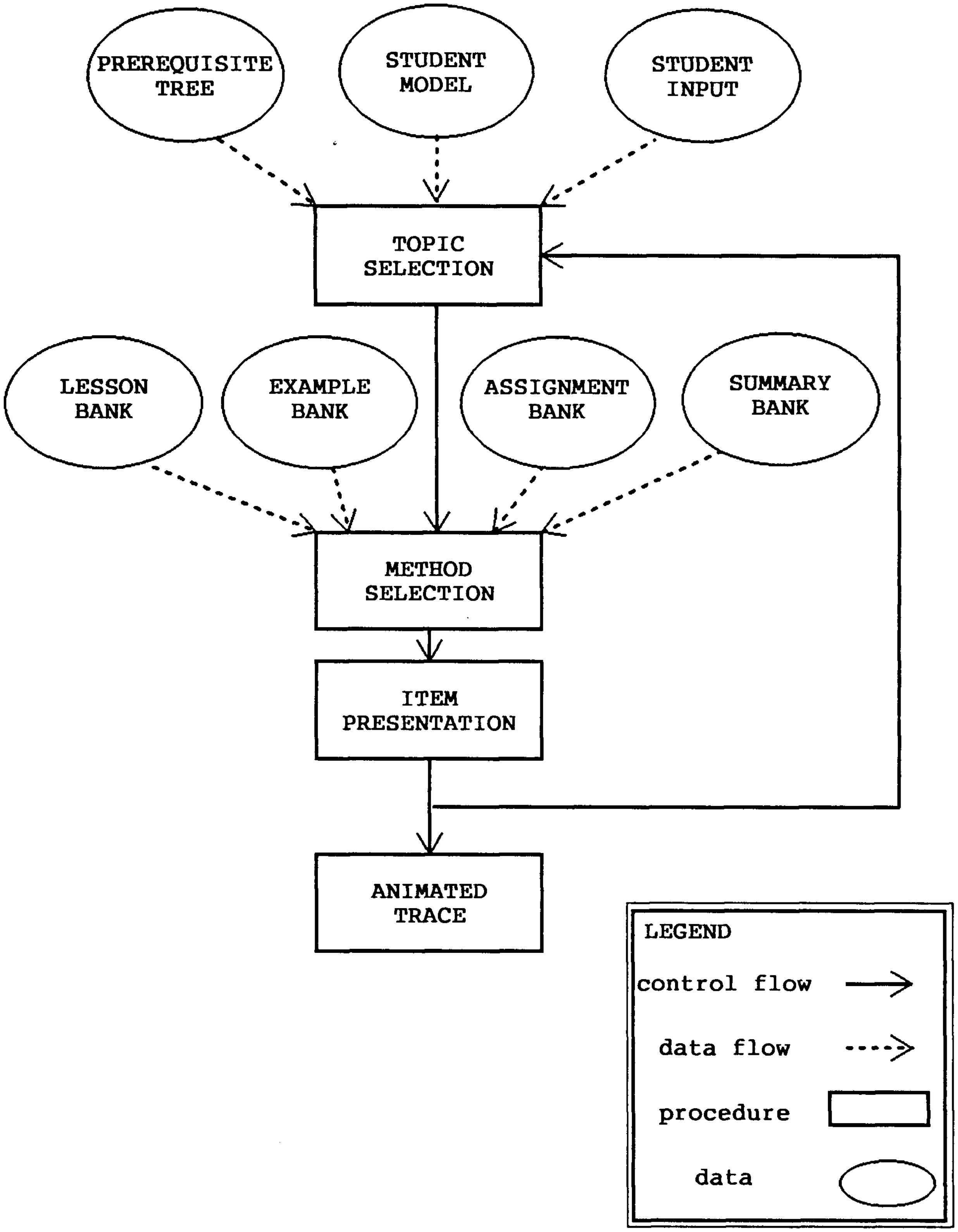


Figure 6: Prolog CAI System Design

The figure should be read from the top. A topic is selected using information from the prerequisite tree, the student model and from student input. The student model is built up as the student progresses through the course, and is initially empty.

Once a topic has been selected, the system chooses a method of instruction. This choice depends upon the lessons, examples, assignments, and summaries that are available for that particular topic. One or more items (lessons, examples, etc.) will be presented until the topic has been satisfactorily completed.

The student model is then updated, and the process repeats.

Students may use the Animated Trace to further investigate many of the examples from the course.

3.3 Methods of Instruction

The system teaches by reference to a bank of instructional materials, which are divided into five categories: lessons, examples, assignments, summaries, and anything else.

Lessons present new material on a given topic. A lesson consists of one or more pages of text. The student can scroll back and forth within a lesson, or suspend it in order to go into Prolog, or to look at an example.

Examples may be small Prolog programs, or merely syntactically correct uses of a built-in predicate. Students can look at the examples, they can go into Prolog to try them out, and they can use the Animated Trace to see how they work.

Assignments consist of one or more short answer or multiple choice questions. An assignment is complete when all of its questions have been correctly answered. To determine if an answer is correct, the system compares it with a set of previously stored answers. It does not attempt to evaluate the correctness of student-written programs, but students can examine these themselves using the Animated Trace.

Summaries are short versions of lessons, used for review and to determine if a student is already familiar with a topic.

Anything Else means instructional materials that do not fit easily into one of the other groups. In general, these may consist of an arbitrary Prolog predicate (for example, a call to a natural language tutoring program). This category was used for the on-line evaluation questionnaire, discussed in Chapter Five.

Each topic may have any number of instructional modules available, in any of these groups. There may, for example, be several examples for a particular topic, or several lessons using different teaching strategies.

3.4 The Prerequisite Structure

To teach a topic, the system will normally first find and teach all of its prerequisites. It finds them by looking at the prerequisite structure.

The representation used for this prerequisite information is the And/Or Tree (McCalla et al., 1982). A diagram showing part of such a tree is given in Chapter One. A topic may have several prerequisites, all required, or it may require only one of a group of prerequisites. This is realized in the And/Or Tree as follows. If a node has several descendents connected by an AND arc, then all are required. If an OR arc is used, then any one of the prerequisites will do.

A topic with AND prerequisites can be taught only after all of its prerequisites have been taught, while a topic with OR prerequisites may be taught after any one of its prerequisites is completed.

McCalla's use of the And/Or tree works quite well, but it does have one problem. To see what that is, we will have to look more closely at the OR node.

The meaning of an OR is that there are several different ways of satisfying a prerequisite requirement. In what circumstances does this actually occur?

In the most common case, there are several different methods of instruction for the same topic (eg. analogy vs. learning by doing). Any one of the methods should result in the same knowledge being learned by the student.

In the other case (much less common), there are two separate bodies of knowledge, either one of which is an acceptable prerequisite to some further concept. For example, the prerequisite to a computer languages course might be a knowledge of any two computer languages.

In McCalla's And/Or Trees, no distinction is made between alternative methods of teaching a single topic, and alternative topics which are each acceptable prerequisites to some third topic. Unfortunately, the two cases are not identical, and should be treated differently.

Consider the choice of method. A good teacher, or a good CAI program, can keep track of how well students cope with different methods of instruction, and can use that knowledge to choose the methods which are most likely to succeed with each student.

The choice of topic is more difficult. It might be done with a shortest path algorithm. The topic chosen would be the one with the fewest prerequisites, so as to fulfil the prerequisite requirements as quickly as possible. On the other hand, perhaps it should be left up to the students, since it depends upon their prior knowledge of Prolog, and their individual interests.

Since the choice of topic differs from the choice of method, the two are separated in the Prolog CAI system. An And/Or tree is used for the topics, and the choice of method is made later on.

Mixing the choice of topic and the choice of method of instruction is not confined to McCalla's system. Goldstein's Genetic Graph, for example, also mixes the two.

3.5 Choosing a Topic of Instruction

The system uses a recursive depth first search of the prerequisite tree to choose a topic of instruction.

The search begins at the root of the tree (the end of the course). If this node has no prerequisites, then it can be taught immediately. If it has AND prerequisites, then each of these must be taught first, along with their prerequisites. If it has OR prerequisites instead, then only one of these need be taught, along with its prerequisites.

Eventually, the search reaches the leaves of the tree (those topics without prerequisites). The path that the search has taken through the tree is one possible path that a student can take through the course. Beginning with the leaf nodes, these topics are presented to the student, and, as each topic is completed, the student follows the search path back towards the root of the tree.

If the student fails to learn a topic, the system will back up and look for an alternative path through the tree by trying other branches at OR nodes. If no better results are achieved on any of the alternative paths, the system returns to the failed topic (in the hope that the student has learned something in the interim, and may be able to succeed where once he or she had failed).

The student does not need to go along with the system's choice of what to study. A student who is bored with a topic can tell the system to look for another one (proceeding as if the current topic had been successfully completed). A student who is having trouble with a topic can ask the system to look for alternatives, or can review one or more of its prerequisites. Finally, a student who wants to guide his or her own studies can disregard all the system's choices, and pick each topic for himself or herself.

3.6 Choosing a Method of Instruction

Associated with each topic in the course are one or more methods of instruction (lessons, examples, assignments and summaries). Once a topic has been chosen, the Prolog CAI system creates a menu listing all of the methods of instruction for that topic (showing them in the same order in which they appear in the Prolog database), and presents that menu to the student. The student can pick any desired method, or he or she can let the system choose.

The system chooses a method of instruction as follows:

Standard Order:

In general, the system will present items in the order in which they appear in the menu. Normally, lessons appear first, followed by examples, assignments, and summaries. This may be changed for any topic by varying the order in which these items are listed in the Prolog database.

Multiple Entries:

In some cases, several lessons exist for a single topic. These are alternative methods for studying the same topic, and so only one of them must be taught. It is only if the student feels the need for another approach that the others will be taught. This would be true also for assignments and summaries as well, but as the course currently stands, no topic has more than one assignment or summary.

3.7 Deciding Which Topics are Previously Known

Often, a student already knows some of the course material, or finds it to be so self-evident that it might as well be known ahead of time.

A CAI system should be able to determine quickly which sections of the course a student already knows, and then use that information in choosing topics for individual students to study. At the same time, its belief that something is known might turn out to be unfounded, so that any topics that are skipped because of it are prime candidates for review if a student has trouble later on.

The Prolog CAI system leaves the decision of what to skip up to the student. The student can ask to leave any topic that seems unnecessary. Any such unfinished topics are included later on if the system is asked to find topics to review.

3.8 Notes on Instructional Methodology

There is no one best way to teach. Human teachers have a wide variety of styles, and so do CAI programs. The structure of the Prolog CAI system allows for a wide range of teaching styles to be used for individual topics.

The Prerequisite Outline, the Topic List, and the list of instructional modules for each topic are stored in a rule-base. This makes them easy to modify during course development, or later, during maintenance.

Lessons, examples and summaries are no more than files of text, which can be easily changed or augmented. Assignments are lists of questions, each followed by the accepted responses, and by the action to be taken, given each response.

The procedures for changing the course (adding material, or re-arranging or revising old material) is described in the Appendices.

3.9 The Implementation

The Prolog ICAI program was written entirely in Prolog. While this made some of the program's features especially easy to implement (such as the creation and traversal of the prerequisite tree), it posed certain problems as well.

CProlog (version 1.1) does not provide any graphical predicates, nor does it allow a Prolog program to make system calls, nor does it allow a Prolog program to call routines written in other languages. The Graphical Trace was intended to be an integral part of the ICAI program, but these deficiencies of CProlog made this impossible. The availability of graphics from within CProlog would also have improved the menu presentation used in the Prolog ICAI program.

Another problem with the implementation turned out to be the inadequacy of ordinary CRT displays for showing large quantities of text. A number of students indicated that they would rather have had the lesson texts on paper than on the screen. This situation will be ameliorated with the use of high resolution bit-mapped workstations with windowing systems.

Chapter Four

Animated Trace Design

4.1 Introduction

This chapter describes the design of the Prolog Animated Trace Program called Anilog (for Animation of Logic).

Most trace programs are sequential. Their output consists of line after line of text, in a terse format that usually omits important information such as assignments of values to variables and the creation and use of data structures. The inclusion of this additional information to a sequential trace makes the output bulky and difficult to follow.

Nevertheless, such information can be very useful in understanding programs, and it is often used by human instructors in the classroom, using such visual aids as flow charts, pointers to program listings, data structure diagrams, system organization charts and so on.

Anilog teaches Prolog in much the same way that a human instructor might. It shows the current goal, along with the relevant parts of the Prolog database, and it points to points of interest as it describes what is happening.

Anilog lies in between Dionne's Antics and Lakin's Pam. It is not a full-fledged programming language. One cannot, for example, write over part of the displayed program and have that change incorporated in the running Prolog program. On the other hand, it is an interactive trace; it can execute

Prolog programs and (to some extent) it can control their execution.

4.2 General Design

There are two parts to Anilog. The first is an interactive Prolog trace program, which produces voluminous sequential output. The second part is a graphical display program, which reads the output from the trace and displays it in a compact graphical format on the terminal screen.

The two programs are intended to run concurrently, so that all of the interactive trace options are available along with the graphical display. When this is not possible (as happened in the implementation), then the two programs can be run in sequence. One can run the trace interactively, and then graphically display the results.

The two parts to Anilog will be described separately below.

A short sample protocol for Anilog is given at the end of Chapter One, and a more complete example may be found in Appendix 4.

4.3 The Trace

The standard Prolog trace programs do not produce enough information about Prolog's database searches and about backtracking.

A new trace program was written, which does produce all the necessary information - including such things as the database clauses that Prolog tries to use to unify with the current subgoal, and the variable bindings which result.

The trace is completely interactive. The user can step through the program, stopping at any of the entry, re-entry, success exit, and failure exit points for each goal. The user can jump over some goals, and can re-direct the program's execution by forcing goals to succeed or to fail.

When run separately from the graphical display program, trace output is written both to the terminal and to a file for later graphical display.

4.4 The Animated Display

The animated trace appears on the terminal screen, with information displayed in three windows - one for the current goal, one for the database, and the last for output produced by the program being traced.

The goal window shows the current goal and the level of recursion. The portion of the goal that Prolog is currently working on (the current subgoal) is highlighted, and messages are produced describing the evaluation of the goal. This window shows, for example, whether the current subgoal succeeds or fails; it shows backtracking; and the instantiation of variable values.

The database window shows the clauses that Prolog accesses while trying to satisfy the goal. Each clause that Prolog tries to unify with a subgoal is displayed; the current clause is highlighted; and messages are produced to report whether unification succeeds or fails.

The user window is used to separate any output produced by the program being traced from output produced by the trace program itself.

When a new level of goal is created (due to the successful unification of a subgoal with the left-hand side of an implication clause), a new goal window is created, and is overlayed on top of the previous one. When this goal is completed, its window is removed, and work resumes on the previous goal, which is now uncovered.

The display produced by the animated trace has been slowed down to suit the average user. When it is not possible to run the animated display concurrently with the interactive trace, the animated display will pause periodically and wait for the user to press <return>. The frequency of these stops can be controlled by setting the leashing mode (to full, half, or unleashed).

4.5 The Implementation

The interactive trace portion of Anilog was written using CProlog (version 1.1). The graphical display portion was written in 'C', using the CURSES window graphics package, which in turn uses the UNIX Termcap terminal capability database.

It was initially expected that the trace output could be piped to the graphical display, allowing both programs to run concurrently. However, when this was tried, none of the trace output was passed to the graphical display program until after the end of a CProlog session. Therefore, the Animated Trace is not properly interactive. This situation would not have occurred with a more fully functioned version of Prolog. Both Quintus Prolog and MProlog, for example, provide the user with an external language interface, allowing the direct use from Prolog of the necessary graphical primitives.

The Animated Trace as implemented, correctly traces and displays the execution of a wide variety of small Prolog programs. For some larger programs, however, problems appeared. When the program to be traced exceeded 10 levels of recursion, or overflowed windows, information was occasionally written to the wrong part of the screen.

It is not clear to what extent this was due to bugs in the Animated Trace, and how much it was due to problems with CURSES, and/or with Termcap.

Chapter Five

Evaluation and Conclusions

This chapter describes the procedures used to evaluate the Prolog CAI System and the Animated Trace, discusses the results of this evaluation, and presents some conclusions.

5.1 Prolog ICAI System: Evaluation Procedure

Two evaluation procedures are built into the CAI system.

The first is an on-line comments facility. At any time during the course, a student can write comments on the course. These are stored in a file which can later be edited and mailed to a system maintenance person.

The other built-in evaluation procedure is an on-line questionnaire. After students have completed a few topics they are asked to answer some questions about the course. The questionnaire comes after the student has gained some familiarity with the way the program works, but early enough that he or she will still remember any difficulties in learning to use any of its features.

Users who did not complete the questionnaire on-line were asked to complete it by hand.

The questionnaire is reproduced on the following page.

In addition to the on-line evaluation procedures, I talked informally with each of the system's users. I sat beside some of them as they were actually using it, in order to see first hand the problems that came up.

Prolog CAI Course Evaluation

1. How much Prolog did you know before you started this course?

2. Which of the options (described in help menus) have you tried? Which do you never use, and why?

3. How often do you chose topics for yourself, instead of letting the system choose?

4. Is the material in the course presented at the right level for you?

5. Have you used the Table of Contents, the Prerequisite Outline, or both? How useful are they, and how could they be improved?

6. Is there anything you would like to be able to do, but can't?

7. Would you rather not see the Topic Menus, and go directly into lessons, etc.?

8. Are the Help messages useful?

9. How does this system compare with any other CAI systems you have used, or with a Prolog text, or a professor?

Figure 7: Prolog CAI Evaluation Questionnaire

About a dozen people participated in the evaluation. Five of these were computer science graduate students and Professors; the rest had little or no previous experience with computers.

All of the users were thrown at the system with very little preparation. Three had previous experience with Prolog, but the rest knew only that they would be taught a language called Prolog. They were not given any advance description of the system, or of Prolog.

5.2 Animated Trace: Evaluation Procedure

In his paper, "What Stories should we tell Prolog Students", Alan Bundy (1983) considers the advantages and the disadvantages of six different methods of demonstrating Prolog's proof procedure. He lists 10 ideals for a Prolog story, and these will be used to evaluate the Animated Trace.

<u>The Ideal Prolog Story</u>

1. The overall search space of the call would be conveyed; in particular, the backtracking points would be indicated, and it would be obvious when ultimate success has been attained.
2. The flow of control through the search space would be indicated.
3. Each subgoal literal would be displayed.
4. The clauses that resolve it away would be displayed.
5. The unifiers produced by these resolutions would be displayed.
6. The remaining literals would be displayed.
7. The other clauses that could resolve with the selected literal would be displayed.
8. The final instantiation of the original goal would be displayed.
9. Different instantiations of a clause would be distinguished.
10. The effect of a cut on the search space would be indicated.

In addition, a Prolog story should not be so cluttered with information as to be unreadable. Bundy notes that all of the stories he studied can be extended to cover the above points, but doing so would leave them too cluttered to be useful for all but the simplest of problems.

5.3 Prolog ICAI System: Evaluation Results

Most of the naive users were content to type <return> and let the system guide their studies. Occasionally, one would choose a topic from the Topic List, but for the most part, they did not try out most of the options available to them.

The more advanced users experimented with more of the options, and most found them useful. The only thing that was regarded as unnecessary was the Prerequisite Outline (which was seen as duplicating the facilities of the Topic List). One complaint was that there was nothing in the system to guide the users in their choice of topics. Since that is precisely the goal of the Prerequisite Outline (showing the structure of the course), clearly it was not being as helpful as was intended.

Students who knew no Prolog thought the course material was at an appropriate level for them, and everyone who used the help messages liked them (with reservations noted below).

The use of a mixed-initiative dialogue worked out well. Naive users tended to leave most choices up to the system, while more experienced users (particularly those with some Prolog background) made most decisions for themselves.

There was a series of complaints about reading text from a terminal screen. The program's users should have been provided with a higher quality paper copy of all of the program's lessons.

Several students had trouble going back to review specific points from earlier on in the course. They would know what they wanted to review, but would not know exactly where to find it.

In general the system proved to be useful, but it cannot stand alone. Everyone who participated in the evaluation agreed that it was useful for learning Prolog. Most, however, had some trouble in learning how to use the system.

Sophisticated users had the fewest problems. They were used to learning the ins and outs of new computer systems, and had little trouble adjusting to this one.

Naive users, on the other hand, had a great deal of difficulty learning to use the system unaided. The particular problems varied from individual to individual, but at some point each needed to be helped along.

While they liked the system's help messages, the naive users wanted more. They would have preferred a natural language explanation of just what was happening, and of what was expected of them at any time. Naive users were often not familiar with the idea that there can be several ways of using a system (or modes), with different actions expected at each.

They would, for example, try to address Prolog from within the CAI system without first calling CProlog, or they would try to choose a Topic List command, when they were in the middle of a lesson describing the Topic List.

Students often misjudged their own capacity to absorb new material. They would read several lessons in a row, and then go into Prolog to try many things at once. By this time, however, they had forgotten some of the material they had just covered, and had to return to the tutorial to try to find it again.

To facilitate this review process, the program should allow users to scroll backwards through the course, in much the same way one can flip back through a book. As it stands, the Prolog CAI Course lets the user scroll backwards within a lesson, but not from one lesson or topic to another.

The course material itself could be more interesting. The things that people liked most were the examples and the Animated Trace (see below). In particular students asked for more interactive examples. They liked having a database which they could query and change, and they liked using the trace to study how Prolog responded to their queries and changes.

One goal for the Prolog CAI system was to be flexible, and it is. New topics can be added; old ones can be re-arranged. Lessons, examples, assignments and summaries can likewise be added or changed.

The system could even be used to teach topics completely unrelated to Prolog, by using an appropriate prerequisite tree, and by writing new instructional modules. Most of the program's facilities would carry over unchanged, although the ability to escape directly into Prolog would only be useful if you were teaching about something that could be shown from Prolog - like the Prolog debug package, or a text editor that could be called from Prolog.

5.4 The Animated Trace: Evaluation Results

The Animated Trace satisfies most of Bundy's criteria for a good Prolog story.

Backtracking is shown; it is obvious when ultimate success (or failure) is reached; the flow of control is displayed; each subgoal literal is shown, along with the clauses that resolve it away; the unifiers produced are shown, as are the remaining literals; clauses not in the final solution path are still shown when they are tried during a proof; the final instantiation of the original goal is displayed; and the effect of a cut is shown, not so much on the search space, but on the movement of the proof procedure through a goal.

The Animated Trace (as implemented) partially fails points 7 and 9. It does not show clauses unless they are tried during a proof, and it does not show the instantiations of database clauses. There is, however, no conceptual reason why it could not do both of these things, and indeed it would be a fairly simple task to include them.

The main failing of the Animated Trace relates to showing the entire search space. It shows that portion of the search space which is traversed during a proof, including blind alleys, but it has no way of showing the remainder (things that might have been).

This means it falls short of the ideal for points 7 and 10 as well. Clauses which could resolve with the selected literal are shown only when they are tried during a proof. When they are not tried, they are not shown. Similarly, the effect of a cut is only shown on that portion of the search space which is investigated during a proof.

When altered to cover points 7 and 9, the Animated Trace comes nearer to the "ideal Prolog story" than do any of the stories Bundy describes. Furthermore, it displays its information in a more concise and more easily followed format than they do.

5.5 Conclusions

In this research I have tried to show how Computer Assisted Instruction can be improved through the use of Artificial Intelligence techniques, and good teaching stories.

The Prolog ICAI program and the Prolog Animated Trace were designed to be flexible and easy to use. These goals have been successfully met.

The And/Or Prerequisite Tree proved to be a natural way to represent the structure of the domain. As added benefits, this structure was easily created and searched, and it helped make the course flexible and easy to change.

The Prolog CAI program uses a very simple student model -- a subset of the Prerequisite Tree. While simple, this student model proved to be entirely adequate. Other, more authoritarian, CAI programs require highly complex student models to reduce their chances of making bad decisions. The Prolog CAI program can make do with a simple student model because it has the student's help with every decision it makes.

The Animated Trace worked especially well in demonstrating Prolog's proof procedure. Students could use it both to study examples from the CAI course and to follow the execution of their own programs. By using graphics it gives more information than traditional Prolog trace programs, without swamping the student with details.

Future versions of the Animated Trace should be more fully interactive. Ideally, students should be able to change the database, rewrite part of the goal, or alter the flow of execution while the trace is in progress. The problems that were encountered in this research were largely the result of inadequate tools. CProlog version 1.1 is entirely lacking in access to the graphical primitives necessary for the Animated Trace, while the poor quality of text on the display made it a chore for students to read through all the lessons. Future users of the CAI program should be provided with a high resolution workstation.

Aside from its lack of graphics, Prolog proved to be a nice language for the implementation. The construction and traversal of the prerequisite tree was particularly simple in Prolog, as was the coding of a modified Prolog interpreter which provided information needed by the Animated Trace.

The Prolog CAI program's lessons cover most of the features of CProlog, but the treatment is at times sparse. More work needs to be done to revise and expand upon the lesson material, and to provide more examples and summaries. A smoother method of paging forward and back through the course material should be implemented, and the display of the Prerequisite Tree should be improved.

Overall though, the programs have shown that wedding Artificial Intelligence techniques with good teaching stories can be used to improve Computer Aided Instruction.

References

[1] Barr, A. and Feigenbaum, E. eds (1982), The Handbook of Artificial Intelligence, Vol. II, Heuristech Press, Stanford California.

[2] Borning, Alan (1979), Thinglab - A Constraint Oriented Simulation Laboratory, Stanford Computer Science Report, STAN-CS-79-749, Standord University, California.

[3] Bundy, Alan (1983), What Stories Should We Tell Prolog Students?, DAI working paper #156, University of Edinburgh, Edinburgh, Scotland.

[4] Burton, R.R., Rubinstein, R. and Brown, J.S. (1976), A Reactive Learning Environment for Computer-Assisted Learning, Bolt, Beranek, and Neuman, BBN report #3314, Cambridge Mass.

[5] Burton, R.R. and Brown, J.S. (1977), A Paradigmatic Example of an AI Instructional System, First International Conference on Applied Systems Research, New York.

[6] Carbonell, J.S. (1970), Mixed-initiative Man-Computer Instructional Dialogues, Bolt Beranek, and Neuman, Technical Report.

[7] Clancey, W. (1979), Tutoring Rules for Guiding a Case Method Dialogue, In Intelligent Tutoring Systems (Sleeman, D. and Brown, J.S. eds), pp. 201-225, Academic Press, London, U.K.

[8] Clancey, W. (1987), Methodology for Building an Intelligent Tutoring System, In Artificial Intelligence and Instruction (Kearsley, G. ed.), Chapter 9, Addison-Wesley.

[9] Clocksin, W.F., and Mellish, C.S., (1981), Programming in Prolog, Springer-Verlag, Berlin-Heidelberg-New York.

[10] Colbourn, M. (1982), An Expert System for the Diagnosis of Reading Difficulties, Proceedings of Expert Systems, Engham, England.

[11] Curran, W.S. (1982), A Teacher/Learner, SIGCSE Bulletin, Vol. 14, No. 1, pp. 229-231.

[12] Crawford, S. (1981), A Standards Guide for the Authoring of Instructional Software, Joint Education Management (JEM) Research, Victoria, British Columbia.

[13] Dageforde, M. and Beard, M.H. (1978), The BASIC Instructional Program Supervisor's Manual, ERIC Educational Database, microfiche #2902.

[14] Dionne, M.S. and Mackworth, A.K. (1978), ANTICS: A System for Animating LISP Programs, Computer Graphics and Image Processing, Vol. 7, No. 1.

[15] diSessa, A. (1986), Principles for the Design of an Integrated Computational Environment for Education, in Children in an Information Age (Sendov, B. and Stanchev I. eds.), Pergamon Press, Oxford, U.K.

[16] Fine, G. (1980), The Design of an Intelligent Lisp CAI Tutor, M.Sc. Thesis, University of British Columbia, Vancouver, B.C.

[17] Fogel, E. (1982), Computers and Education, Honours Research Project, Carleton University, Ottawa, Ontario.

[18] Fogel, E. (1983), A Guide to Intelligent Computer Assisted Instruction, Class essay, CPSC 522, University of British Columbia, Vancouver, B.C.

[19] Forman, D. (1982), Search of the Literature, in Instructional Use of Microcomputers, British Columbia Ministry of Education, Victoria, B.C.

[20] Goldstein, I. (1979), The Genetic Graph: A Representation for the Evolution of Proceedural Knowledge, International Journal of Man-Machine Studies, Vol 11, No. 1, pp. 51-77.

[21] Hofstetter, F.T. (1981), Using the PLATO System, ECCO Newsletter, Vol. 2, No. 3, pp. 23-32, Educational Computing Organization of Ontario, Ontario.

[22] Johnson, L. and Soloway, E. (1987) Proust, in Artificial Intelligence and Instruction, (Kearsley, G. ed), Chapter 3, Addison-Wesley.

[23] Jones, M. ed. (1986), Computational Intelligence, Special Issue: AI Approaches to Education, Vol. 2, No. 2, National Research Council of Canada, Canada.

[24] Kearsley, G. ed. (1987), Artificial Intelligence and Instruction, Addison-Wesley, 1987.

[25] Kimball, R. (1973), A Self-Improving Tutor for Symbolic Integration, Intelligent Tutoring Systems, (Sleeman, D. and Brown, J.S. eds), pp. 201-225, Academic Press, London, U.K.

[26] Kowalski, R. (1979), Artificial Intelligence Series: Logic for Problem Solving, North Holland.

[27] Laubsch, J. and Eisenstadt, M. (1981) Domain Specific Debugging Aids for Novice Programmers, IJCAI-81, pp. 962-969.

[28] Lakin, F. (1980), Computing with Text-Graphic Forms, Proceedings of the Lisp Conference at Stanford University, Stanford, California.

[29] London B. and Clancey, W. (1982), Plan Recognition Strategies in Student Modelling: Prediction and Description, Stanford University Technical Report, STAN-CS-82-909, Stanford, California.

[30] Looi, C. (1984), Explorations of Programming Learning Behavior of Novices through Computer Aided Learning, M. Sc. Thesis, University of British Columbia, Vancouver, B.C.

[31] Matz, M. (1982), Towards a Process Model for High School Algebra Errors, in Intelligent Tutoring Systems, (Sleeman, D. and Brown, J.S. eds), pp. 201-225, Academic Press, London, U.K.

[32] McCalla, G., Peachey, D. and Ward, B. (1982), An Architecture for the Design of Large Scale Intelligent Teaching Systems, 1982.

[33] McCalla, G., Bunt, R. and Harms, J. (1986), The Design of the Scent Automated Advisor, in Computational Intelligence, Vol 2., No. 2, National Research Council of Canada, Canada.

[34] O'Shea, T. (1979), Self-Improving Teaching Systems, in Intelligent Tutoring Systems, (Sleeman, D. and Brown, J.S. eds), pp. 201-225, Academic Press, London, U.K.

[35] Papert, S. (1980), Mindstorms: Children, Computers and Ideas, Basic Books, New York.

[36] Peachey, D. (1982), An Architecture for Plan-Based Computer Assisted Instruction, M.Sc. Thesis, University of Saskatchewan, Saskatoon, Saskatchewan.

[37] Ragsdale, R. (1982), Computers in the Schools: a Guide for Planning, Ontario Institute for Studies in Education Press, Toronto, Ontario.

[38] Searle, J.R. (1980), Minds, Brains, and Programs, The Behavioral and Brain Sciences, Vol. 3, pp. 417-422, Cambridge University Press, Cambridge, England.

[39] Self, J. (1974), Student Models in Computer-Aided Instruction, International Journal of Man-Machine Studies, Vol. 6, pp. 261-276.

[40] Sleeman, D. and Brown, J.S. (1982), Intelligent Tutoring Systems, Academic Press, London, U.K.

[41] Soloway, E., Woolf, B., Rubin, E. and Barth, P. (1983), MENO-II: An Intelligent Tutoring System for Novice Programmers, IJCAI-81, pp. 975-977.

[42] Soloway, E., Woolf, B., Rubin, E., Barth, P., Bonar, J. and Erlich, K. (1981), Why your Students Write Those Crazy Programs, Proceedings of the National Educational Computing Conference, Texas, USA.

[43] Stephens, A., Collins, A. and Goldin, S.E. (1982), Misconceptions in Students' Understanding, in Intelligent Tutoring Systems, (Sleeman, D. and Brown, J.S. eds), pp. 201-225, Academic Press, London, U.K.

[44] Suppes, P. (1972), Computer Assisted Instruction, in Display Use for Man-Machine Dialog (Handler, W. and Weizenbaum, J. eds), Munich, West Germany.

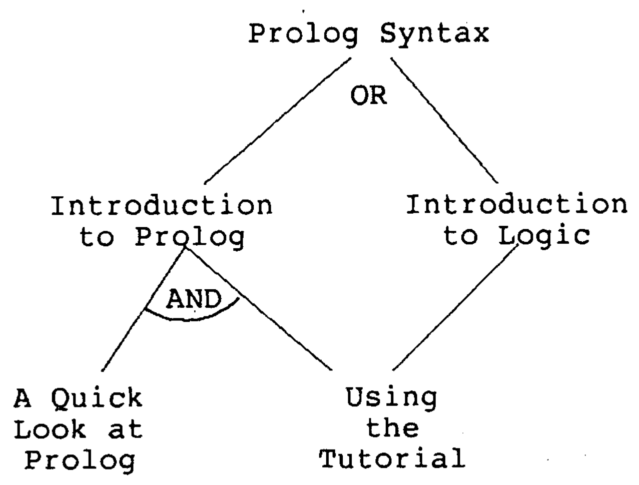
[45] Weizenbaum, J. (1965), ELIZA-A Computer Program for the Study of Natural Language Communication between Man and Machine, Communications of the ACM, Vol. 9, pp. 36-45.

[46] Yazdani, M. (1984), New Horizons in Educational Computing, John Wiley & Sons, Rexdale, Ontario.

APPENDIX 1 - The And/Or Prerequisite Tree

A portion of this tree is shown graphically, then the internal representation for that portion of the tree is given. Finally, the entire tree is given, in the same manner that it is shown to students using the course.

And/Or Tree: Graphical Representation



And/Or Tree: Internal Representation

```
prereq('A Quick Look at Prolog', 'Introduction to Prolog').
prereq('Using This Tutorial', 'Introduction to Prolog').
prereq('Using This Tutorial', 'Introduction to Logic').

orprereq('Introduction to Prolog', 'Syntax').
orprereq('Introduction to Logic', 'Syntax')
```

And/Or Tree: User's View

```
Prolog Basics
  Proof Procedure
    Unification
      Semantics
      - Introduction to Prolog
          A Quick Look at Prolog
          Using This Tutorial
      - Introduction to Logic
          Using this Tutorial
  Side-effects
    Syntax
    - Introduction to Prolog
    - Introduction to Logic
Built-in Predicates
  Input/Output
    Side-effects
    File Access
    Character I/O
    Term I/O
    Reading-in Programs
  Arithmetic
    Operators
      Syntax
  Convenience
  Control of Execution
  The Cut
    Control of Execution
  Comparison of Terms
  Meta-Logical
  Debugging
  Sets
  Program Information
  Environment
  Changing the Data Base
  Internal Data Base
Definite Clause Grammars
  Syntax
```

Each topic is shown with its prerequisites indented beneath it. 'Or' prerequisites are indicated with a preceeding minus sign. For conciseness, a topic's prerequisites are only shown once, the first time that the topic appears.

APPENDIX 2 - Prolog ICAI User's Manual

STARTING UP

Assuming the CAI system is in a file called 'caifile', type:

```
CProlog         (Call CProlog from the shell)
[caifile].      (Load the system)
cai.            (Remember the period!)
lines(N).       (where N is the number of lines on your terminal)
                If N = 60, you can skip this.
                If N < 40, things won't fit on the screen.
```

LEAVING THE SYSTEM

Type either:

```
CProlog       (puts you in CProlog)
save          (saves what you've done, then puts you in CProlog)
```

LEAVING CPROLOG

Type 'cai.' to resume the tutorial.

Type 'halt.' to quit.

Remember the P.E.R.I.O.D.S.

RESTORING SAVED STATES

Suppose the state is in file 'oldstate'. Next time you start CProlog, type: CProlog oldstate

COMMENTS, GRIPES, ETC.

Mail them to me.

From within the system, type 'comment'. Everything that you type from then on, until an end-of-file (↑D or ↑C), will be put into a file called 'mailtofogel'.

You can figure out what to do with that yourself.

APPENDIX 3 - Anilog Users Manual

STARTING UP

Anilog's input is the output from a special CProlog trace program. If that is in a file called 'traceout':

Type: anilog traceout

USING ANILOG

At times, Anilog pauses to let you think about what you are seeing. Here is what you can type at these times.

```
<cr>        - continue the trace
quit        - abort the trace
full        - set leashing to full (pause more often)
half        - set leashing to half (pause occasionally)
unleash     - turn off leashing (never pause for input)
```

BUGS

Anilog is not robust, and is not guaranteed to work on arbitrary Prolog goals. When it bombs, the terminal may be in an unusual state. To get it back to normal, type:

```
<linefeed>  (NOT return)
reset       (NOT the reset key, type the letters  r e s e t)
<linefeed>
```

MAKING NEW INPUT FILES

Load the file ~fogel/cai/trace/trace into CProlog.
Call mytrace(G), where G is the goal to be traced.

```
eg.         ?- mytrace( (write(hi), write(hi)) ).
```

Output will be displayed on the terminal, and will also be written to a file named 'traceout' in your current directory.

APPENDIX 4 - Animated Trace Example

The Example Program and Query:

```
----------------------------------------
|  i_like(rice).                        |
|  i_like(Thing) :- has_fur(Thing).     |
|  i_like(Thing) :- can_walk(Thing),    |
|                   can_talk(Thing).    |
|                                       |
|  has_fur(dog).                        |
|                                       |
|  can_walk(X) :- person(X).            |
|                                       |
|  can_talk(radio).                     |
|  can_talk(X) :- person(X).            |
|                                       |
|  person(marc).                        |
|                                       |
|                                       |
|  ?- i_like(Y), has_fur(Y), write(Y).  |
----------------------------------------
```

```
                         Prolog Animated Trace

Current Goal:  (Level  1)
---------------------------------------------
|i_like(Y),                                 |
|has_fur(Y),                                |
|write(Y).                                  |
|                                           |
|                                           |
|                                           |
---------------------------------------------

Prolog Database:
--------------------------------------------------------
|                                                      |
|                                                      |
|                                                      |
|                                                      |
|                                                      |
--------------------------------------------------------

User Output:
--------------------------------------------------------
|                                                      |
|                                                      |
|                                                      |
|                                                      |
|                                                      |
--------------------------------------------------------
```

```
                         Prolog Animated Trace

Current Goal:  (Level  1)
---------------------------------------------------
|i_like(Y),    <-- current subgoal                |
|has_fur(Y),                                      |
|write(Y).                                        |
|                                                 |
|                                                 |
|                                                 |
---------------------------------------------------

Prolog Database:
----------------------------------------------------------------
|i_like(rice).  <- try to unify with subgoal                   |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------

User Output:
----------------------------------------------------------------
|                                                              |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------
```

```
                          Prolog Animated Trace

Current Goal:  (Level  1)
---------------------------------------------
|i_like(Y),   <-- current subgoal            |
|has_fur(Y),                                 |
|write(Y).                                   |
|                                            |
|                                            |
|                                            |
---------------------------------------------


Prolog Database:
---------------------------------------------------------
|i_like(rice).  <- unified                               |
|                                                        |
|                                                        |
|                                                        |
|                                                        |
---------------------------------------------------------


User Output:
---------------------------------------------------------
|                                                        |
|                                                        |
|                                                        |
|                                                        |
|                                                        |
---------------------------------------------------------
```

```
                        Prolog Animated Trace

Current Goal:  (Level  1)
--------------------------------------------------
|i_like(rice),   <-- success                      |
|has_fur(Y),                                      |
|write(Y).                                        |
|                                                 |
|                                                 |
|                                                 |
--------------------------------------------------

Prolog Database:
---------------------------------------------------------------
|                                                             |
|                                                             |
|                                                             |
|                                                             |
|                                                             |
---------------------------------------------------------------

User Output:
---------------------------------------------------------------
|                                                             |
|                                                             |
|                                                             |
|                                                             |
|                                                             |
---------------------------------------------------------------
```

```
                    Prolog Animated Trace

Current Goal:  (Level  1)
-----------------------------------------------------
|i_like(rice),                                      |
|has_fur(rice),   <-- current subgoal               |
|write(Y).                                          |
|                                                   |
|                                                   |
|                                                   |
-----------------------------------------------------

Prolog Database:
------------------------------------------------------------------
|has_fur(dog).     <-- try to unify with subgoal                 |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
------------------------------------------------------------------

User Output:
------------------------------------------------------------------
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
------------------------------------------------------------------
```

Prolog Animated Trace

```
Current Goal:  (Level  1)
---------------------------------------------------
|i_like(rice),                                    |
|has_fur(rice),    <-- current subgoal            |
|write(Y).                                        |
|                                                 |
|                                                 |
|                                                 |
---------------------------------------------------

Prolog Database:
----------------------------------------------------------------
|has_fur(dog).     <-- not unified                             |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------

User Output:
----------------------------------------------------------------
|                                                              |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------
```

```
                        Prolog Animated Trace

Current Goal:  (Level  1)
---------------------------------------------------
|i_like(rice),                                    |
|has_fur(rice),    <-- failed                     |
|write(Y).                                        |
|                                                 |
|                                                 |
|                                                 |
---------------------------------------------------


Prolog Database:
----------------------------------------------------------------
|                                                              |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------


User Output:
----------------------------------------------------------------
|                                                              |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------
```

```
                    Prolog Animated Trace

Current Goal:  (Level  1)
--------------------------------------------------
|i_like(Y),   <-- redo subgoal                    |
|has_fur(Y),                                      |
|write(Y).                                        |
|                                                 |
|                                                 |
|                                                 |
--------------------------------------------------

Prolog Database:
---------------------------------------------------------------
|i_like(rice).                                                |
|i_like(Thing) :- has_fur(Thing).  <-- try to unify with subgoal|
|                                                             |
|                                                             |
|                                                             |
---------------------------------------------------------------

User Output:
---------------------------------------------------------------
|                                                             |
|                                                             |
|                                                             |
|                                                             |
|                                                             |
---------------------------------------------------------------
```

```
                         Prolog Animated Trace

Current Goal:  (Level  1)
---------------------------------------------------
|i_like(Y),    <-- redo subgoal                   |
|has_fur(Y),                                      |
|write(Y).                                        |
|                                                 |
|                                                 |
|                                                 |
---------------------------------------------------

Prolog Database:
----------------------------------------------------------------
|i_like(rice).                                                 |
|i_like(Thing) :- has_fur(Thing).    <-- unified               |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------

User Output:
----------------------------------------------------------------
|                                                              |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------
```

```
                        Prolog Animated Trace

Current Goal:  (Level  1)
---------------------------------------------------
|Current Goal:  (Level  2)                        |
|-------------------------------------------------|
||has_fur(Thing).   <-- current subgoal         ||
||                                              ||
||                                              ||
||                                              ||
---------------------------------------------------


Prolog Database:
----------------------------------------------------------------
|has_fur(dog).     <-- try to unify with subgoal               |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------


User Output:
----------------------------------------------------------------
|                                                              |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------
```

```
                          Prolog Animated Trace

Current Goal:  (Level  1)
---------------------------------------------------
|Current Goal:  (Level  2)                        |
|-------------------------------------------------|
||has_fur(Thing).   <-- current subgoal          ||
||                                               ||
||                                               ||
||                                               ||
---------------------------------------------------

Prolog Database:
----------------------------------------------------------------
|has_fur(dog).     <-- unified                                 |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------

User Output:
----------------------------------------------------------------
|                                                              |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------
```

```
                        Prolog Animated Trace

Current Goal:  (Level  1)
----------------------------------------------------
|Current Goal:  (Level  2)                         |
|--------------------------------------------------|
||has_fur(dog).      <-- Goal succeeds            ||
||                                                ||
||                                                ||
||                                                ||
----------------------------------------------------

Prolog Database:
-----------------------------------------------------------------
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
-----------------------------------------------------------------

User Output:
-----------------------------------------------------------------
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
-----------------------------------------------------------------
```

Prolog Animated Trace

```
Current Goal:  (Level  1)
--------------------------------------------------
|i_like(dog),      <-- success                   |
|has_fur(Y),                                     |
|write(Y).                                       |
|                                                |
|                                                |
|                                                |
--------------------------------------------------


Prolog Database:
----------------------------------------------------------------
|                                                              |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------


User Output:
----------------------------------------------------------------
|                                                              |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------
```

```
                    Prolog Animated Trace

Current Goal:  (Level  1)
-----------------------------------------------
|i_like(dog),                                 |
|has_fur(dog),     <-- current subgoal        |
|write(Y).                                    |
|                                             |
|                                             |
|                                             |
-----------------------------------------------

Prolog Database:
-----------------------------------------------------------
|has_fur(dog).     <-- try to unify with subgoal           |
|                                                          |
|                                                          |
|                                                          |
|                                                          |
-----------------------------------------------------------

User Output:
-----------------------------------------------------------
|                                                         |
|                                                         |
|                                                         |
|                                                         |
|                                                         |
-----------------------------------------------------------
```

```
                          Prolog Animated Trace

Current Goal:  (Level  1)
---------------------------------------------------
|i_like(dog),                                     |
|has_fur(dog),     <-- current subgoal            |
|write(Y).                                        |
|                                                 |
|                                                 |
|                                                 |
---------------------------------------------------

Prolog Database:
----------------------------------------------------------------
|has_fur(dog).     <-- unified                                 |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------

User Output:
----------------------------------------------------------------
|                                                              |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------
```

```
                    Prolog Animated Trace

Current Goal:  (Level  1)
----------------------------------------------------
|i_like(dog),                                      |
|has_fur(dog),      <-- success                    |
|write(Y).                                         |
|                                                  |
|                                                  |
|                                                  |
----------------------------------------------------

Prolog Database:
-----------------------------------------------------------------
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
-----------------------------------------------------------------

User Output:
-----------------------------------------------------------------
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
-----------------------------------------------------------------
```

```
                    Prolog Animated Trace

Current Goal:  (Level  1)
---------------------------------------------------
|i_like(dog),                                     |
|has_fur(dog),                                    |
|write(dog).       <-- current subgoal            |
|                                                 |
|                                                 |
|                                                 |
---------------------------------------------------

Prolog Database:
----------------------------------------------------------------
|                                                              |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------

User Output:
----------------------------------------------------------------
|                                                              |
|                                                              |
|                                                              |
|                                                              |
|                                                              |
----------------------------------------------------------------
```

```
                    Prolog Animated Trace

Current Goal:  (Level  1)
----------------------------------------------------
|i_like(dog),                                      |
|has_fur(dog),                                     |
|write(dog).         <-- Built-in Succeeds         |
|                                                  |
|                                                  |
|                                                  |
----------------------------------------------------


Prolog Database:
-----------------------------------------------------------------
|                                                               |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
-----------------------------------------------------------------


User Output:
-----------------------------------------------------------------
| dog                                                           |
|                                                               |
|                                                               |
|                                                               |
|                                                               |
-----------------------------------------------------------------
```

```
                    Prolog Animated Trace

Current Goal:  (Level  1)
----------------------------------------
|i_like(dog),                          |
|has_fur(dog),                         |
|write(dog).      <-- Goal succeeds    |
|                                      |
|                                      |
|                                      |
----------------------------------------

Prolog Database:
---------------------------------------------------
|                                                 |
|                                                 |
|                                                 |
|                                                 |
|                                                 |
---------------------------------------------------

User Output:
---------------------------------------------------
| dog                                             |
|                                                 |
|                                                 |
|                                                 |
|                                                 |
---------------------------------------------------
```