Design Considerations

of an

Intelligent Tutoring System

for

Programming Languages.

Mark T. Elsom - Cook.

Submitted in fulfillment of the requirements
of the degree of Doctor of Philosophy, to

University of Warwick,
Coventry CV4 7AL.

Research conducted in the Department of Psychology.

October 1984

CONTENTS

Figures
=======

SUMMARY

The overall goal of the thesis is to attempt to highlight the major topics which must be considered in the design of any Intelligent Tutoring System and to illustrate their application within the particular domain of LISP programming.

There are two major sections to the thesis. The first considers the background to the educational application of computers. It examines possible roles for the computer, explores the relationship between education theory and computer-based teaching, and identifies some important links among existing Tutoring Systems. The section concludes with a summary of the design goals which an Intelligent Tutoring System should attempt to fulfill.

The second section applies the design goals to the production of an Intelligent Tutoring System for programming languages. It devises a formal semantic description for programming languages and illustrates its application to tutoring. A method for modelling the learning process is introduced. Some techniques for maintaining a structured tutoring interaction are described.

The work is set within the methodology of Artificial Intelligence research. Although a fully implemented tutoring system is not described, all features discussed are implemented as short programs intended to demonstrate the feasibility of the approach taken.

To George,

for

unfailing advice

and

Kate,

for

unfailing friendship.

I would like to express my thanks to the following people who have helped me during the course of writing this thesis;

George Kiss: my supervisor, for guiding me through the discovery of Artificial Intelligence, and for ensuring that Warwick Psychology Department has the facilities to support such work.

Mark Steedman, John Pickering: for variously inspiring me and picking the rubbish out of my mind.

Gerry Kelleher, Ian Carr, Roy Pinder: for commenting upon earlier versions of the thesis, discussing various aspects of the system, and generally being useful.

Tony Cohn, Edmund Furse: for comments on parts of the thesis.

Alison Petrie-Brown: For reading more drafts than the mind can comfortably conceive, and providing encouragement at times of need.

All the M.Sc students who have been on the receiving end of my programs, and still managed to produce constructive criticism.

Various members of the departments of Computer Science, Education, Philosophy and Psychology at Warwick, who introduced me to a number of unfamiliar topics.

Kate Stainton-Ellis: For useful discussions, helpful advice, and generally keeping the show on the road.

Finally, I would like to express my gratitude to my parents, without whom ...

" The cardinal function of the teacher, in the early stages, is to get the pupil on the inside of the form of thought or awareness with which he is concerned. At a later stage, when the pupil has built into his mind both the concepts and mode of exploration involved, the difference between teacher and taught is obviously only one of degree. For both are participating in the shared experience of exploring a common world. The teacher is simply more familiar with its contours and more skilled in finding and cutting pathways. The good teacher is a guide who helps others to dispense with his services."

[Peters 1966]

Part 1

An Overview.

Chapter 1.

Overview of thesis.

1.-Origins of thesis.

This project, like much recent work in the Educational Application of computers, derives its basic motivation from the work conducted at M.I.T. on teaching LOGO to children.

The literature from this LOGO group suggests that the computer is a powerful tool for tackling educational problems. It is suggested that letting a child interact freely with an appropriate computer-based environment is sufficient to permit the pupil to discover great things for herself.

In the spirit of this LOGO approach I attempted to design a friendly environment (MATILDA) in which postgraduate students could learn LISP by discovery [Elsom-Cook 1982] . In essence, by producing an environment in which every error a student made resulted in constructive feedback, MATILDA attempted to emulate the LOGO experience. The system has been used practically for several years, and it was immediately apparent that simply placing a student in front of a computer, no matter how friendly the environment, was not sufficient to ensure that she learnt LISP. Worksheets, individual tuition and other forms of guidance were necessary.

Although this programming environment did not achieve its original aim, the limitations suggested a possible method for extending the system. Since it is generally the case that a learning task involves interaction with an environment and with a teacher, MATILDA provided a starting point for an investigation of the nature of the "missing" component - guidance. The work reported in this thesis stems from that investigation.

2.-Overview of the thesis.

The purpose of this thesis is to investigate the nature of the teaching interaction. A system intended to guide a discovery learning interaction about the programming language LISP will be described, and the issues of representation of domain knowledge, skills of interaction and knowledge about the user will be discussed.

2.1-Methodology.

The study is set within the framework of producing software capable of acting as a stand-alone tutor for the programming language LISP. A tutoring system called IMPART, which is an instantiation of a more general tutoring system architecture will be described. IMPART is intended to support the early stages of learning a programming language, and to facilitate easy transfer to normal programming environments once programs become too complex for it to make a reasonable teaching contribution.

This work embodies a specific model of the teaching interaction. Let us suppose that a teacher has her own (usable) representation of a domain of knowledge, and that the pupil also has a representation of knowledge which may be of relevance to the domain. The goal of teaching must be to provide the pupil with a model of the domain (at least) as powerful as that of the teacher. This model will not necessarily have the same internal form, since it must be bound to differing cognitive structures in the two individuals, but both models must be consistent with the real world. A "direct teaching" strategy involves the teacher in translating her model into a suitable external representation. The learner then attempts to integrate features of this representation with her preexisting cognitive structures, and possibly to modify the form of these structures. One way of categorising varieties of teaching interaction is by locating the point in this "transfer of knowledge" at which major reorganisation of the knowledge occurs. The following examples will illustrate this;

In a traditional "chalk and talk" environment, the teacher generates an external form from her domain knowledge which is reasonably independent of the states of knowledge of the pupils whom she is teaching - at best it is based on a simple "prototypical" student model. This leaves the student to do all the work of adapting the knowledge for her own understanding. A further instance of a situation in which the burden of adapting knowledge rests on the student alone occurs when attempting to learn from a book.

At the opposite extreme, the tutor devotes much energy to developing an accurate model of her pupil. She then combines this model with her domain knowledge to generate an individualised course of tuition. In this approach, the tutor must constantly update her model and reassess the presentation method. A fairly complete exposition of this approach to education will be found in the work of Jean-Jacques Rousseau [Rousseau 1762] . Rousseau's philosophy is based on providing one teacher per child who tries to model the child's interests and readiness to learn, and ensure that the appropriate experiences occur in the environment to trigger each phase of learning. It should be noted that it is actually possible to exert complete control over the pupil by this method, while maintaining the impression of freedom. Rousseau's own ideas of what is "natural" have a great influence on what the pupil is allowed to learn.

Between these extremes there is a continuum of teaching styles. The above framework permits a distinction to be made between "good" and "bad" teachers and "good" and "bad" learners. A good teacher is one who successfully restructures information to make it easier to learn before presenting it. A good learner is one who is capable of doing large amounts of reorganisation of knowledge to make it fit into her existing cognitive structures, and of modifying those structures when necessary. We can see that good learners are less dependent on the ability of their teachers than are poor learners.

If we assume that the pupil has the clearest idea of her current state of knowledge and that the teacher has the clearest understanding of the domain, then it is reasonable to suppose that an optimal teaching interaction is one in which the matter to be presented and the form of it's presentation is decided by negotiation between the two parties. From this it follows that a study of teaching should seek to identify the issues about which negotiation may take place, and should investigate the form of that negotiation.

2.1.1-Implications for user modelling.

An examination of the teaching techniques applied in Intelligent Tutoring systems to date shows that they concentrate on producing a copy of the teacher's model of the domain in the head of the pupil. While this technique is the most straightforward to study, it is a dangerous practical tool since it has a strong tendency to produce uniformity among the pupils to whom it is applied. It is also apparent that, unless much extra work is done by the pupil in integrating this knowledge with her existing structures, her model cannot be more powerful than that of the teacher. These techniques remain close to the "chalk and talk" methods of teaching.

In its most restrictive form this approach is manifest in Computer Aided Learning packages which make little attempt to adapt to the user. It is also apparent in Intelligent systems which use overlay modelling (discussed in chapter 5); the user model is built as a subset of a model of an expert in the problem domain. The process of instructing the pupil involves finding discrepancies between the two models and attempting to remove them. These systems often use direct testing strategies to find evidence for the existence of units of the model. This should not be confused with systems which maintain a descriptive representation of an "expert" level of achievement. This is necessary for assessment of the pupil, but the details of the experts approach are not used in tutoring. In essence, this suggests that the teacher's expert knowledge of the domain should be regarded as a black box, and any glass box form which the pupil sees should be adapted to that individual.

An improvement of this type of model is to explicitly include representations of variations on the expert skill which are not actually used by the expert, but which reflect likely deviations from the model such as simplified versions of complex rules. The "bug" representation of systems such as BUGGY [Burton 1982] are examples of this category. This provides a better means of categorising the difficulties of a pupil, but does not affect the basic problem since these systems still have a goal of making the pupil into a copy of the expert.

In IMPART, an attempt has been made to avoid this restrictive aspect of the interaction. The system maintains a descriptive model of expert competence in programming, and tries to detect discrepancies between this and a descriptive model of the pupil's programming. When an inconsistency is detected, the system brings it to the attention of the pupil so that the pupil may make appropriate corrections to her model. No particular model of expert problem-solving ability or bugs is explicitly maintained in the system for the purpose of tutoring. The pupil model is derived by applying a model of the learning process to the same information sources which the pupil is able to access.

2.1.2-Educational position.

Most Intelligent Tutoring systems make use of a highly constrained interaction such as Socratic tutoring [Collins 1980] . More recent systems begin to permit mixed-initiative interaction with a greater degree of freedom for the pupil, but they are still found to revert to their restrictive form in difficult situations in order to tie people to the model which they know about. The method adopted in designing IMPART is to approach the problem from the other end of this Freedom-Constraint dimension; the system leaves control of the interaction to the pupil, and only interferes when it has a justification for doing so.

The Educational Philosophy of "Discovery Learning" on which the child-centred education movement (and recent computer applications such as LOGO at MIT) has been based raises major issues about the degree and type of guidance which should be given to a pupil. In particular, it is generally acknowledged that totally "free" learning, in which a learner is simply left alone in an environment, is a very poor use of available resources, and unlikely to lead to high levels of attainment [Dearden 1967] . It is interesting to note that although LOGO is often thought of as a free environment for learning, a case study of LOGO use at MIT [Solomon 1976] shows an interaction which is actually tightly constrained in practical terms, and in which the pupil exercises virtually no control.

It is apparent that discovery learning offers advantages in terms of the motivation of the individual. If we consider the form of teaching which is necessary in "guided discovery" learning, we find that we can reduce the amount of detail involved in negotiating the course of teaching, since much of the learner's contribution can be in terms of actions in the environment. This reduces the amount of direct interaction needed between teacher and pupil, since the role of the teacher is now assessing the possible directions which the pupil could follow at a given point, and encouraging those which she considers to be most productive. This leaves the major problem of assessing the likely value of a course of action.

It was decided to design a system which provides an environment in which a pupil can "discover" the language Lisp. The system also contains a teaching program which monitors the interaction of pupil and environment, and attempts to make positive contributions to that interaction. The system has a goal of detecting skills which the user is ready to learn, and encouraging the exploration of those skills, either by manipulating the environment or by making direct "teaching statements" to the pupil.

3.-Structure of thesis.

The thesis is divided into four sections, together with some appendices.

Part 1 (this section), provides a brief overview of the thesis and the tutoring system which it describes. It also attempts to clarify the possible roles which a computer can play in the educational process.

Part 2 discusses the basic issues in education and tutoring system design which determined the nature of IMPART.

Part 3 is the central component of the thesis. It describes the tutoring system in detail, and shows how the general issues which have been discussed may be practically applied.

Part 4 summarizes the major points which the thesis has made.

The appendices list some of the more interesting parts of the implementation.

The thesis attempts to discuss aspects of Education Theory and Tutoring System design. It is unusual for these areas to appear together in a single document, but their juxtaposition reflects the author's view that it is vitally important to design tutoring systems with reference to a larger educational framework. Both these aspects of tutoring system design require a substantial amount of further research. Rather than attempting to produce a definitive piece of work on either aspect this thesis aims to show the way in which these areas should be related to make Intelligent Tutoring System design a truly interdisciplinary field.

Chapter 2.

Overview of the tutoring system.

1.-The overall structure of IMPART.

Conceptually, the system may be regarded as three separate entities. The first is the PUPIL, which is a system user plus various interfacing packages to enable her to interact smoothly with the system. The second is the ENVIRONMENT, which is the domain about which the pupil is trying to discover together with some tools to aid her exploration. In this case the environment is a LISP interpreter, and the tools are such things as Trace packages and Editors. The third item is the TEACHER, which is considered to embody the intelligent aspects of the system. The teacher monitors the interaction between pupil and world, and attempts to build a model of the state of the pupil which can be used to decide when to intervene in this interaction. It is also possible for the pupil to appeal directly to the teacher for aid.

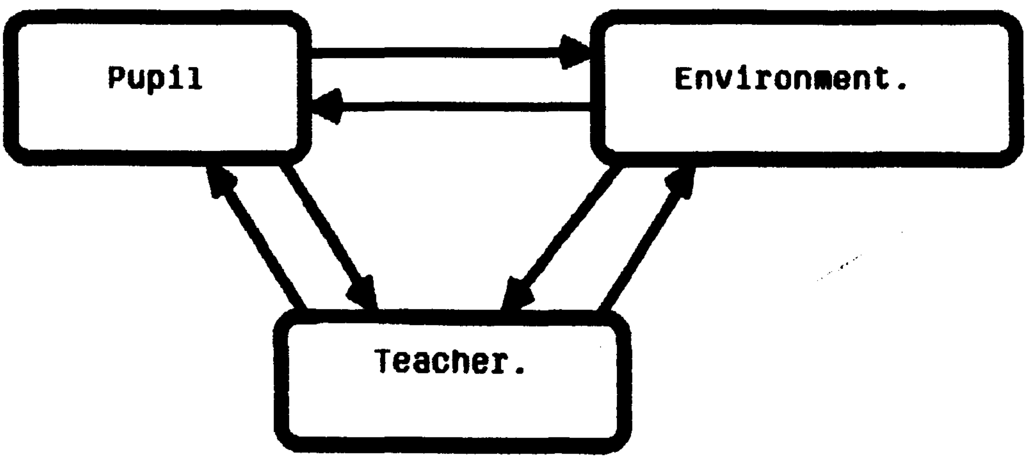


Figure 1 - Guided discovery learning framework.

1.1-An example interaction.

Let us examine the action of the system when dealing with a specific problem. Assume that a task has been agreed between teacher and pupil which is to define a function to extract the second element from a list;

```
(FRED (QUOTE (A B C))) --> B
```

One correct solution would be

```
(DE FRED (X)
   (CAR(CDR X)))
```

and we will assume that the teacher is aware of this solution. Suppose that the pupil offers the following definition;

```
(DE FRED (X)
   (CDR (CAR (QUOTE X))))
```

The tutor executes the pupil's expression before it is completed in order to find possible errors. A violation of the semantics due to an inappropriate argument to CAR (i.e. X) is detected. It finds that there is one evaluation too few on the variable X (by comparing this with the correct solution), and notes that removing QUOTE would correct this. The tutor makes a simple patch to get round this error and continues execution. A second error, due to an inappropriate argument to CDR is found. Comparison with the correct solution detects the difference in evaluation order, so this problem is also noted. It is important to note that, rather than being "the right solution" to the problem, what the system tries to generate is a solution using techniques which are currently at the boundary of the pupil's skill level; the solution form is indirectly determined by the user model.

The comparisons of solutions are not done in the programming language itself, but rather in the underlying semantic representation. This enables the system to identify the reasons for a difference. These differences can be used to identify necessary changes in surface form, or can be discussed in their own right.

The tutor now compares the detected errors with the user model. It finds evidence that this pupil has experience of the multiple-EVAL problem, but has had less experience of the EVAL-order problem. The teaching strategies use this to decide to make a simple statement of fact about the former difficulty, but pass control to a specialized teaching unit (a "topic controller") for a more detailed discussion of EVAL-order.

The EVAL topic controller waits until the problem becomes manifest to the pupil before doing anything. It takes this decision based on the evidence which the user model provides for the pupil's ability to deal with error situations, together with the pupil's experience of this type of problem. The pupil tests the function with '(D C A). An error results and the pupil immediately asks for help. The topic controller attempts to illustrate the error by breaking the problem into sub-problems which make the difficulties more obvious. The tutor asks "WHAT is (CAR (QUOTE (D C A))) ", then "WHAT sort of argument does CDR expect". If this is insufficient to cause the user to generate a correct answer then it uses a more informative technique.

1.2-Knowledge Sources.

An attempt has been made to produce a clear separation between the knowledge sources in the system, so that the examination of the way in which they interact to produce a reasonable teaching sequence is possible. A further goal of this separation was to make the system capable of teaching different programming languages with little overall modification.

1.2.1-Knowledge about the problem domain.

1.2.1.1-Syntax.

In effect, problems of syntax have been bypassed in this system. The pupil works through a syntax-directed editor which is driven from a Backus-Naur form description of the language. The editor generates a parse-tree of any expression in the language, which is the form on which all other parts of the system act. This technique was originally investigated in the EMILY system [Hansen 1971] . The current system was designed for novice programmer's rather than experts, so the constraints upon the editor design were different from those of Emily. In particular, the overhead of learning to use the interface was kept to a minimum.

The boundary between syntax and semantics is not perfectly defined, and some problems are difficult to assign to either class. An example of this is the question of declaring variables before using them. While many systems assume this to be a high level of syntactic information (e.g. [Teitelbaum 1981]), IMPART assigns it to the semantics, because it represents an aspect of the language for which underlying principles exist.

1.2.1.2-Semantics.

An examination of standard techniques of program description showed that although they are well suited to mathematical manipulations, they are not the most appropriate representation for transferring a "psychologically reasonable" model of the language to a pupil. For this reason a different semantic representation was devised. This representation describes each statement in the language in terms of preconditions for it's application plus a body of commands to execute in order to achieve the appropriate effect. The commands are drawn from a set of about fifteen primitive operations which constitute the lowest level of semantic description in the system. This has similarities with certain aspects of Axiomatic and Operational semantics. In choosing this representation, some of the constraints of mathematical consistency and completeness for which other approaches have aimed, have been relaxed. The form used is a type of Direct semantics, so it is unable to handle unusual control flow features such as Jumps or Errors. Other limitations include an inability to prove termination of a program, and some limits involving aliasing of variables.

This declarative form of semantics is acted upon by different programs for each of the applications to which it is put in the system. The most obvious application is execution of expressions in lieu of an interpreter. The system is often able to execute incomplete expressions and ignore those errors which are due to the incompleteness. Describing statements of the language or behaviour of an expression is done by mapping the semantics onto some simple English descriptions of the primitive elements. Automatic generation of problems and problem-solutions using the semantics is an important area, but is was decided that it should not be a major research goal in the design of this system, so it has not been examined in detail. The teaching methodology requires that the action of programs can be made visible in such a way as to avoid implying a particular mechanism, so some effort is devoted to providing the teacher with different ways of displaying information during execution.

1.2.1.3-Higher level structures.

So far, the system has been applied to small domains, where it is possible to detect correct problem solutions by exhaustive search. In real programming domains the search space will be too large to apply this technique, so the system requires the search to be constrained by likely solution forms. More importantly, the student is unlikely to discover such techniques as recursion by trial and error, so some representation of these must exist if the teacher is to guide the pupil towards them. These representations must include conceptual roles (such as "stopping condition") to which actual elements of the program can be assigned. A representation such as that used in the Programmer's Apprentice system [Rich 1981] is envisaged, but it will not be incorporated into the system until the relationship between these structures and lower level programming knowledge has been further investigated.

Some "knowledge" about high level concepts of the language may be thought to reside in the topic controllers of the domain, but this knowledge is not accessible to the reasoning components of the system. In fact, this knowledge is more about conventional methods of describing the units than about their actual semantic form.

1.2.2-Knowledge about the student.

The student model of the system embodies most of the information which the system has about the pupil. Some information about the dialogue state may also be regarded as part of the user model.

The existence of a history of interaction in the system seems important for several reasons. Being able to refer back to recent events in the interaction helps provide a continuity to the teaching, and enables discussion of features which the English interface cannot handle. Exactly which past events can reasonably be referred to is not yet clear, and it is apparent that a principled solution to this problem must involve another level of user modelling. This may lead to a more selective form of history storage than is currently used.

It may prove necessary for the system to do some updating of the user model offline in order to maintain a reasonable interaction rate. What form this processing will take has not been examined, but it is certainly the case that human teachers consider problems which their pupils have presented them outside direct teaching time. It is likely that a teaching system capable of learning from experience would focus on this activity.

Part of the student model is maintained by building a language description corresponding to the language which the student thinks she is using. In the WUMPUS system [Goldstein 1982] , Goldstein identified a set of rules for approaching new knowledge in ways based on previous knowledge. These extrapolation rules identified possible forms of transition between "islands of knowledge" in a "Genetic Graph". IMPART takes a similar approach, but the transitions are generated during the interaction (rather than being wired into the curriculum), and the rules which are used derive from a Machine Learning methodology [Michalski 1983] . A crude monitoring system attempts to assign "success values" to each rule depending on how well the system's goals are achieved if they involve that type of rule. More subtle techniques for exploring and modelling the pupil's own extrapolation rules should be investigated. It is interesting to note that there is a possibility of giving the system a goal of extending the pupil's rule set.

The learning model is not claimed to produce the same hypotheses as the pupil. Rather, it provides an upper and lower bound for the hypotheses which the pupil could have generated at a particular point. These hypotheses can be used as the basis for deductive reasoning mechanisms to predict consequences of these hypotheses. Testing these predictions allows the system to bring the bounds closer together until it has a sufficiently accurate model of the . student (in some particular area) to make a tutoring contribution.

In knowing when to intervene, the system must have some idea of the pupil's ability to handle the problem with which she is currently confronted. In part this is achieved by examining the student's knowledge of the current problem area, but it also involves knowing about the pupil's ability to trace a bug from the sort of feedback which she receives. For this reason, the system should monitor the effectiveness with which pupils can respond to error messages. A goal of the system is to improve this ability so that transfer to normal interpreters is fairly straightforward.

1.2.3-Knowledge about interaction.

Unlike most Intelligent Tutoring systems, which are purely reactive, this system is intended to participate in a structured interaction with the pupil. Much of the work in this section makes use of recent progress in models of conversation [Power 1979] [Reichmann 1978] . Control of the interaction is shared between three major units;

1) Descriptors of domain.

Each concept about which the system is able to talk has a topic controller associated with it. This topic controller embodies all domain specific aspects of the concept, so it contains such things as outlines of various ways to present and discuss a topic. It also contains mechanisms to assess it's own importance at the current point in the interaction, and routines which enable it to make some assessment of it's own success in modifying the user's model of the problem_domain. These mechanisms may contribute to the interaction by passing data to standard teaching strategies, or they may control an interaction specific to their own domain. These controllers may be thought of as including the tutorial goals of the teacher. Each controller can be regarded as an agent, with associated declarative and procedural knowledge.

2) General interaction skills.

These skills have the purpose of maintaining the consistency and smoothness of the interaction. They perform such tasks as marking subject boundaries, and adjudicating in the process of topic selection by choosing a topic controller on the basis of a bid indicating the likely importance and relevance of the topic. Since these procedures make no assumptions about the actual actions of a topic controller, they can integrate linguistic and non-linguistic forms of interaction.

3) Teaching strategies.

Teaching strategies are special conversation techniques which can be called by topic controllers. They are intended to represent those processes of communication which are common to all areas of the domain. They include such things as asking questions, giving examples, and making direct statements of fact.

2.-Implementation of IMPART.

Since the focus of this research was on the design of a tutoring system, the implementation was not carried to completion. Instead, a number of small programs were produced intended to demonstrate the feasibility of the approach. Figure 2 shows the extent of the implementation.

The architecture of IMPART is shown in terms of information flow between components. It is probably best explained by following the passage of information round the diagram, beginning with the pupil.

Our PUPIL may interact with a SYNTAX-DIRECTED EDITOR to produce expressions in the language which she is learning. This editor produces a parse-tree of the language statement which is transmitted to the INTERPRETER. This transmission occurs at all stages of construction of the expression, so that the system may provide information during construction of an expression.

When an expression is complete, and the student wishes to execute it, the EDITOR sends it to the INTERPRETER. The INTERPRETER executes it and returns results and error messages to the editor which displays them in an appropriate form for the PUPIL (e.g. unparsing them). The system described so far corresponds to the passive programming environment in which the PUPIL will ultimately work.

Parsed expressions, with a description of their behaviour, are sent from the INTERPRETER to the LEARNING MODEL, and the PATCHER. The LEARNING MODEL applies learning mechanisms to generate a set of possible hypotheses which the student could draw from the current example. These are passed to an ASSESSOR which compares them with the actual language definition and filters out those which are not valid. The resulting hypotheses are passed, as a maximally-specific/minimally-specific pair, to the CONVERSATION CONTROLLER which stores them as a record of that interaction step.

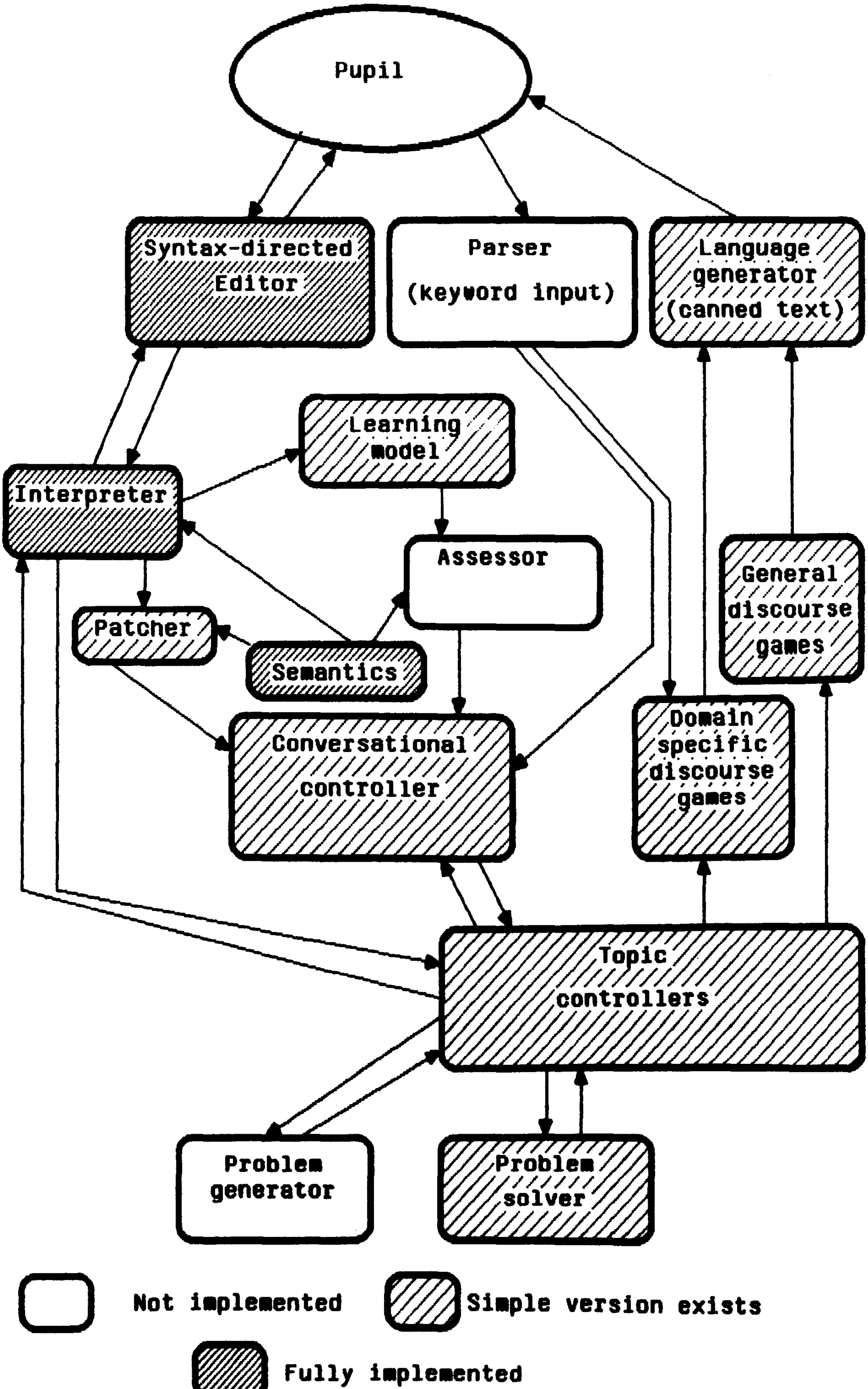
The PATCHER operates on expressions which have produced errors to find specific modifications which would remove the error. It also produces a set which identifies each basic error and the manifestations of that error. This information is passed to the CONVERSATION CONTROLLER.

The CONVERSATION CONTROLLER operates by repeatedly invoking TOPIC CONTROLLERS and enabling them to bid for control of the conversation. It gives these controllers access to knowledge about the current state of the interaction which has been drawn from different sources.

Each TOPIC CONTROLLER may pass information to one or more DISCOURSE GAMES of either type. These GAMES may, in turn, invoke a language generator to present information to the pupil in an appropriate form. In order to access detailed information which it may require to instruct the pupil, each TOPIC CONTROLLER may invoke the INTERPRETER, a PROBLEM GENERATOR and a PROBLEM SOLVER. Each of these units can respond to instructions from the CONTROLLER (for example, the problem generator can be requested to produce a problem involving values of atoms, the interpreter can be asked to reexecute an expression while tracing particular components etc).

Direct interaction between pupil and teacher occurs via the parser (or other input software) which directly affects the conversational context maintained by the CONVERSATION CONTROLLER. This section has not been implemented.

Figure.2 - Implementation overview



3.-A general tutoring system architecture.

The structure of IMPART which is shown in Figure 2 is suitable for tutoring any programming language. To change language it is necessary to provide alternative syntactic and semantic descriptions (in a form which will be discussed later), and probably some additional topic controllers. All other parts of the system operate on the basic semantic description, and hence do not need to be changed for languages other than LISP.

IMPART itself is a particular instantiation of a more general guided discovery architecture. This architecture is shown in figure 3. It will be observed that it focusses on the internal structure of the tutor. It is claimed that this provides a framework within which tutoring systems for a wide range of domains can be constructed. There are two key features; one is the division into teacher, pupil and environment while the second is the central position given to "bounded user modelling" and the model of learning.

The division into three parts distinguishes the active, changing components which constitute the teacher, from the reactive system which the pupil is learning to understand. It is possible to conceive of a degenerate case in which no interaction with the environment occurs, but in general a pupil should apply things which are learnt in some environment.

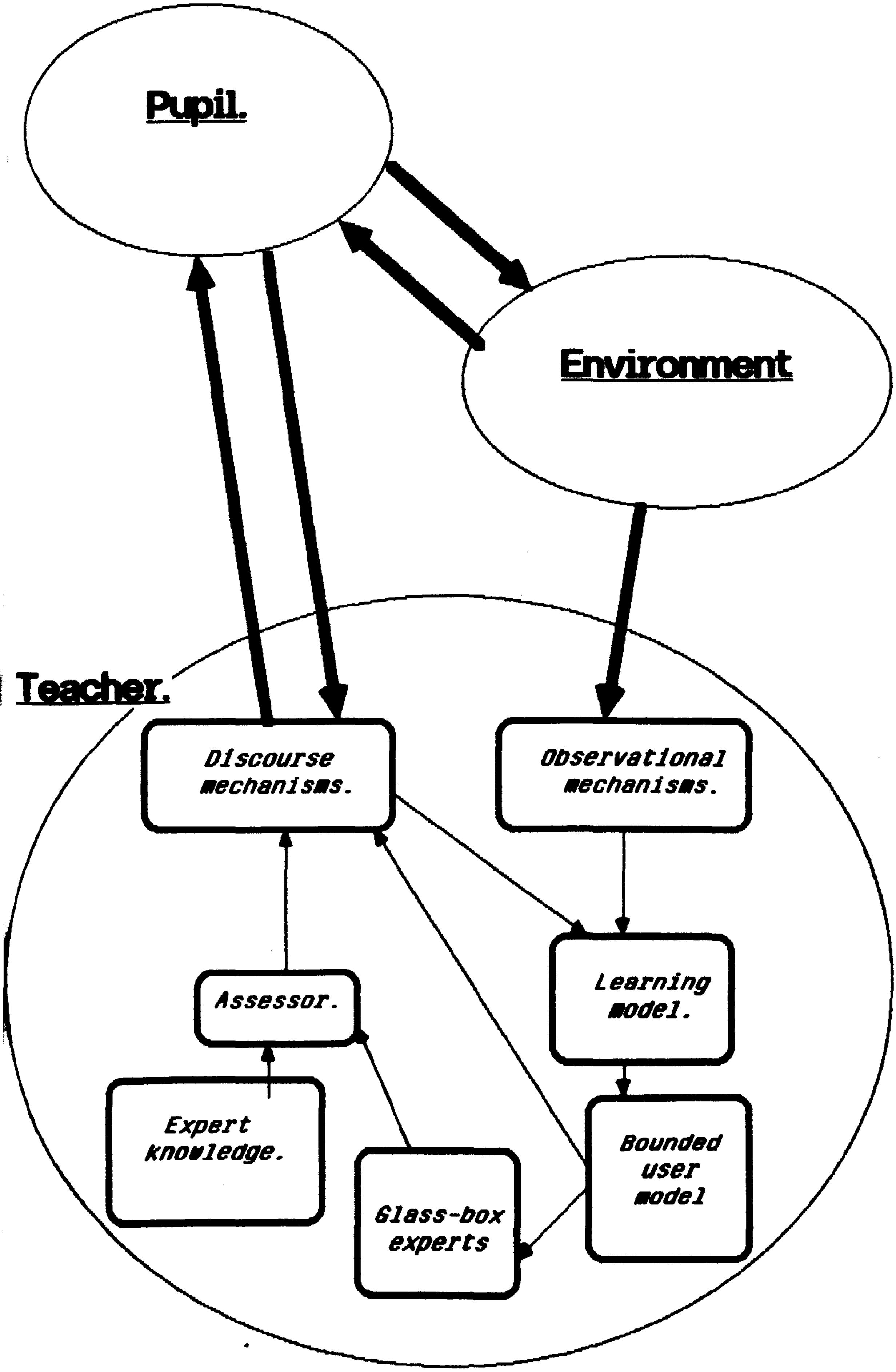
The "bounded user modelling" approach which was described briefly above, provides an alternative to expert-based modelling. Its centrality in this system implies that the focus of the tutor's activity is concerned with building and maintaining a model of the knowledge state of the pupil. An adequate modelling mechanism is a prerequisite of any Intelligent Tutoring System. All tutoring activities are strongly dependent upon the quality of this modelling component.

In figure 3 the tutor interacts with the world via "discourse mechanisms" and "observational mechanisms". The former deal directly with the pupil, the latter with events that occur in the environment. Both these units provide information which is acted upon by the "learning model". This "learning model" generates sets of hypotheses which become the "bounded user model".

The "discourse mechanisms", in addition to an internal model of the discourse structure which controls the interaction, may be affected by the "bounded user model" and the "assessor". The "bounded user model" identifies things which may usefully be pursued from the current conversational state (for example, if atoms and lists have been talked about, the user model would indicate that a discussion of datatypes would be reasonable, but the discourse mechanisms alone would not be aware of this). The "assessor" identifies differences between the performance of an "expert" in the domain, and a "glass-box expert" which attempts to find the consequences which follow from the hypotheses which are currently in the "bounded user model". The "glass-box expert" is expected to work with techniques which are on the current bounds of the pupil's understanding, but is not to be regarded as a model of the problem-solving skills of the student; it is an approximation to such a model, and should always be treated as an approximation. The "discourse mechanisms" may directly invoke the "glass-box expert" to generate and solve problems.

In an actual implementation it is difficult to separate each of the components of the general tutoring system architecture. In IMPART, for example, the observational mechanisms include the syntax-directed editor and the semantically-driven language interpreter, but these units also play roles as part of the environment, and (to some extent) as part of the black-box expert.

Figure 3. - A general tutoring system structure (Bounded user modelling).



Chapter 3.

Computers in an educational environment.

1.-The role of computers in Education.

The computer is having a major effect upon many aspects of modern life. It is a central focus of the technological revolution which we are currently facing.

There seems to be a certain degree of inevitability in the way that computers appear in all types of environment, so much so that they are often accepted unquestioningly as a sign of progress in a particular area.

1.1-Computers as an educational tool.

Within the Educational system the problems are particularly apparent. Computers seem to be a vital part of every secondary curriculum, primarily in terms of "computer awareness" (teaching people what computers can do), but also in the role of aids to teaching. These trends are spreading to primary and infant schooling.

There are two extreme positions which educationalists may hold with respect to the issue of computers in education. On one side are those who advocate wholesale acceptance of this technology while on the other are those who simply ignore it in the hope that it will go away. It is remarkable that a high percentage of teachers adopt one or the other of these extreme positions, rather than possessing opinions which lie between them. Let us examine these points more closely.

The group which advocate acceptance of the computer into all aspects of education typically do so without good reason. A computer is considered to be appropriate because it is high technology, and therefore progress. In taking this attitude we are ignoring a fundamental tenet of any educational system; Each new idea should be thoroughly tested and examined, and introduced ONLY if it appropriate to do so. There is no merit in taking drill and practice arithmetic from a book and placing it on a computer screen - nothing has been added. Computers free machine-based programmed learning from the burden of expensive task-specific hardware which was a major obstacle to this movement. Unfortunately, this has the effect of eliminating the practical problems of the approach without resolving the theoretical ones.

Another practical difficulty for this approach is that it typically talks about the value of computers per se; there is no real appreciation that the versatility of the computer is such that assessment should be carried out upon each piece of software individually.

The teachers who reject computers altogether as an educational tool are probably taking a sensible decision given the information which is available to them. Most teachers have only seen the output of typical commercial software houses.

The problems with these programs are many. A large number of software companies with experience in fields other than education produce Educational Software. It is only recently that such companies have begun to consult educationalists about the design of their programs. Much of the early work made use of the primitive facilities which a computer offers (such as iteration) to produce very mechanistic software (such as rote learning programs) which was economical in machine time, but of dubious educational value. These programs often repeated the trends followed by the movement towards programmed learning in Education which occured during the 1960's. It is small wonder that teachers who see computers in this light dismiss them as simply a more technological version of this unsucessful paradigm. A classic dismissal of this type is that given by H.Broudy [Broudy 1969] . In this paper the author points out that the most important element of a Socratic Dialogue was Socrates. If a machine cannot support the level of reasoning which Socrates must have used during a discourse then it can only be a sham, simply giving the impression of a useful dialogue.

Unfortunately, the recent initiative to put "microcomputers in schools" placed the emphasis on providing hardware without considering the problems of educating teachers about the potential of these machines. As a result, both the views given above have become more entrenched - there are more computers sitting unused, and more alienated members of the teaching profession than there were before.

There is a need to educate teachers about the facilities which properly written software could offer. Until such information is made available it is unreasonable to expect teachers to be able to decide about the applicability of the computer to various educational tasks. I would propose that there are two main themes to discuss; the first is identifying Educational tasks which can be usefully tackled by existing technology, and the second is creating an awareness of the new potential which Artificial Intelligence based techniques could bring to education.

1.1.1-Applications for existing technology.

A major problem with existing Educational software is that it does not adapt to the individual user, and as such produces a stereotyped interaction. The dangers of non-adaptive systems will be discussed later. For the present, we may note that the behaviour of a group of children is much more predictable than that of an individual. It would seem reasonable, therefore, to propose that current software methods could be justifiably applied to the production of programs intended to promote interaction among a group of children. Anyone who has watched a group of children interacting with a computer will know how much argument can be produced by something as simple as a hangman program. The ability to communicate ideas to your peers is important. If computers can help to achieve this then we have found a role for them which is not adequately covered by any other (currently practicable) method. Within this area, I suggest there exist two major applications for the computer.

The first application is that of a "Quiz-master". By this I mean that the computer should present problems to a group of pupils which are intended to fire debate among them. These questions would be provided in advance by a teacher as they would be for programmed learning software. The difference is that the focus is not upon the answers given, but upon the discussion which precedes those answers. The computer would not be aware of such discussion, or capable of contributing to it.

The unique facility offered by the computer in this case is the prospect of Immediate Feedback on answers (in a way which a book could not provide), without inhibiting the discussion in the way that the presence of a teacher would. It is important to note that correct use of such software is not simply a matter of picking half a dozen pupils at random and sitting them in front of the machine. The teacher is required to use her judgement to set up groups who are likely to interact well together. This highlights a general difficulty with current approaches to using computers in education; it is wrong to attempt to use them in isolation from other activities - they should be integrated with the curriculum and become an extension to the teachers repertoire of tools rather than a substitute teacher.

The second role which I would propose is that of "Computer as Pupil". It is acknowledged that attempting to teach something is a powerful method for clarifying and structuring that information. If pupils habitually taught the subject that they are learning it would provide an outlet through which to polish communicative skills, while also causing the pupil to think about the material in such a way as to bring about deeper understanding. There are certain practical difficulties in using this technique in schools. The teacher cannot be a very convincing pupil since such a role conflicts with the image of a teacher as a source of knowledge. Recent Educational research has examined the provision of this facility in the form of peer-group teaching [Furlong 1976] . It is certainly the case that children impart information to their peers, but the spirit of competition prevents a child's own class from being a good audience. Another problem is that with all the children trying to teach (often teaching the same thing), there aren't enough pupils to go round!

I would suggest that the Computer can alleviate this problem by providing a medium through which pupils may teach. It seems unlikely that current educational software methods could produce a computer which acts as an appropriate pupil since this problem still presents major research difficulties, but it may provide an intermediate device allowing pupils to prepare educational material. The sort of interaction envisaged here is that a group of pupils, having studied a given area, get together to jointly produce a simple Computer Aided Instruction package for that area. Existing author languages (if made slightly more friendly) are perfectly adequate for this sort of work. Whether the end product is ever used is irrelevant, since the educational experience occurs for the designers during the design process.

The idea of pupils teaching each other through the computer is distinct from the idea of learning by telling the computer what to do (an idea associated with LOGO). In this case the pupils are directly concerned with the process of ordering and presenting information to their peers. It directly fosters the development of communication skills.

1.1.2-Artificial Intelligence in Education.

I propose to divide the potential applications for Artificial Intelligence influenced computer use into two categories. The first is to use a computer as an aid to thought in the way that Artificial Intelligence research uses it, while the second is the application of systems which embody the results of Artificial Intelligence research.

In research, the computer acts as a tool to clarify thought processes. Producing a computer program based upon some theory gives a concrete form to the abstract theory. The computer provides a simulation environment which may represent things that are not directly perceivable in the real world. Immediate feedback can be provided on the consistency and consequences of a particular theory.

This view of the computer is at the base of those approaches to computers in education which involve providing a programming environment for the pupil. LOGO [Papert 1971] and PROLOG [Ennals 1981] have been applied within this paradigm. It is important to note that this approach does not centre upon a simple relationship between computer and child - the entire environment in which the child operates is important. Some attempts to introduce such approaches into the classroom have failed because it is not realized that they require a large amount of teacher re-education, and a considerable expenditure of effort in integrating the computer into the educational environment. This reflects the fact that much early research in this area ignored the large scale context of the computer application.

Having said this, programming based approaches are very successful as aids to Artificial Intelligence research, and it is likely that they can be equally constructive in other areas (such as teaching mathematics) PROVIDED APPROPRIATE ENVIRONMENTS ARE DEVELOPED.

Artificial Intelligence research involves the production of computer based models of behaviours which may be regarded as 'intelligent'. It is clear that teaching is such a skill. Research into computer models of teaching involves the combination of many aspects of A.I. research in exploring this very complex skill. If such research produces powerful models of teaching, then it may be possible to apply these models in an Educational setting as teachers.

With the low cost of computers, computer based teaching systems could be provided on an individual basis allowing a level of personalization in education which has only previously been available to the few who could afford personal tutors. A computer could be the constant companion of an individual, never tiring and always focussed upon the needs of that individual. The machine could combine the expert knowledge of many humans (as a book might do) while retaining an adaptiveness to the individual during educational interactions.

The efficacy of this approach to education is very much dependent on the quality of computer-based teaching. While the above system would be of great value if computers were as proficient as good human teachers, anything less could be extremely dangerous. Even if computers achieved the level of poor human teachers, the scope for difficulties such as indoctrination of pupils is great.

At present there are no Intelligent Tutoring systems capable of providing an adequate practical interaction, even in very restricted domains. I wish to propose that this area should provide a central focus for research on computers in education. It may lead towards more powerful computer based teachers, and in the process will serve to clarify the nature of education at a level which may be of practical importance for human teachers.

It may be noted that the two approaches described above may be combined. The pupil may interact with a computer-based environment, with help provided by a computer-based teacher. This may be extremely interesting in that it provides a way to explore those things which the pupil requires over and above a powerful mechanism for representing theories.

1.2-Computers in Educational Research.

Teaching, as was pointed out by Whitehead [Whitehead 1932] , is commonly regarded as an art. It seems unusual to talk about the "Science of Education", and yet such an important topic must be amenable to scientific techniques.

There exists a strong field of empirical educational research, but when attempts are made to attach this work to some theoretical framework the results are often vague and unconvincing. Bennett [Bennett 1976] describes the "...current chaos which is a pretence at research". He points out that

> "...it must be obvious to the critical reader that what is missing from many of the reported studies is the sense of direction and controlled orderliness which can only be provided by an adequate theory."

It is clear from this report that there are difficulties in providing a suitable theoretical framework for educational research. It is not possible to provide a complete and accurate description of a particular teaching method when, as Bennett points out, even the term "teaching method" has not been given a stable definition.

The nature of educational theories is similar to that of psychological theories - indeed, Bruner (among others) has proposed the division of educational theories into educational philosophy and psychological theories. In recent years the development of Artificial Intelligence has introduced the computational metaphor to psychology and provided a powerful formalism for expressing psychological theories. This approach has gained a large following, and computer-based "Cognitive Science" is becoming a major field.

Given that this formalism has had such an impact on Psychology, it seems likely that a similar benefit to Educational Research may be obtained by the introduction of Artificial Intelligence methods. Computer models of the processes of teaching and learning, and the interaction between the two, may provide the basis for a clearer theoretical framework for research on the nature of education.

There has been little interaction between Artificial Intelligence and Education theory as yet. One existing example of such an interdisciplinary approach is the work of Collins [Collins 1982] . While developing a cognitive model of interactive teaching, Collins applied computational metaphors to produce a description of the behaviour of teachers in certain circumstances. This information was used as the basis of a computer-based teaching system which maintained a Socratic dialogue with a pupil. The information collected could alternatively be applied to teacher training - providing a more precise definition of Socratic tutoring. The tutoring system based upon this work will be further discussed in chapter 5.

The work reported in this thesis may be regarded as the beginnings of an attempt to examine Artificial Intelligence based models of the process of education. It seeks to establish a framework within which a variety of models of learning and teaching may be compared.

Summary of Part 1.

This part of the thesis has provided an overview of the thesis itself, and of the tutoring system IMPART. A discussion of the role of computers with respect to education has suggested their use both as a teaching aid and as a means for exploring the nature of the educational process in a rigourous manner. Themes which have been introduced here will be discussed in more detail in subsequent parts of the thesis.

Part 2 -

Background to the system design.

Chapter 4.

Education theory.

1.-Educational background of the system.

In selecting a particular method of teaching, various decisions must be taken about exactly WHAT is being taught and how to teach it. In order to clarify the form of these decisions, the second part of this chapter discusses the nature of theories of Education and outlines some major components of an educational theory suitable for an Intelligent Tutoring system.

An important factor influencing the choice of theory is the model of learning upon which it is based. As Bruner points out [Bruner 1966] ;

> "... a theory of instruction ... must be congruent with the theories of learning and development to which it subscribes."

It is remarkable that many Educational Theories do not explicitly state the Theory of Learning with which they are associated. No Intelligent Tutoring Systems have assigned learning theory a central role in their design. It was such a change as this which Goldstein was requesting when he asked for tutoring system research to move from an "expert-oriented paradigm" to a "learner-oriented paradigm" [Goldstein 1982] .

In order to approach this difficulty, the first part of this chapter attempts to briefly summarize some theories of learning and outline their consequences for Instructional Techniques. The second part discusses the nature of educational theories, and the final part outlines the need for a new theory of education to relate computer-based learning to other educational activities.

1.1-Theories of learning.

A vital component of any theory of education must be a detailed model of the learning process. Unless the mechanisms by which learning occurs can be specified, it is not possible to produce a theory which can describe methods for controlling these mechanisms to achieve our specified ends.

It is difficult to define learning. The problem is that the common usage of the term covers a multitude of activities which occur in a wide variety of situations. Some examples of "learning activities" are; learning to ride a bicycle, learning the names of the kings of England, learning to perform symbolic integration, and learning to maintain relationships with other people. The common features of these tasks are not immediately apparent.

While there have been many interesting attempts to produce detailed theories of learning in recent years [Bower 1981] , there has not been sufficient common ground between them to allow the evolution of a single unified theory of learning. It has been suggested that this may be due to the fact that learning itself is not a unitary phenomenon [Borger 1966] , or it may be that there is a highly general learning mechanism which is manifested in very different ways in different situations. In either case, a consequence is that the few truly general statements which have been made about the nature of learning offer little help in modelling specific learning situations.

In this section we will briefly examine two theories of learning, each of which is of limited scope. We will conclude by outlining an approach to modelling learning which is of particular relevance to the tutoring system outlined in part 3 of the thesis.

1.1.1-Behaviourism.

Behaviourism has been an important area of learning theory for many years.

While most people acknowledge the limitations of behaviourist theories, the approach has been very influential in the past. Since so much of the literature is written with these overtones it is often difficult to avoid slipping into a behaviourist perspective on learning.

The essence of behaviourism is an emphasis on the observable. A learning system receives certain inputs (Stimuli) and it produces certain outputs (Responses). Learning is regarded as the process by which new links are formed between stimulus and response. This does not involve considering any internal "cognitions" of the learning system.

Skinner has provided a framework within which behaviourist methodology could be applied in the Educational process. This involved methods of dividing presentation of material into small units which could be categorized as learnt or not, and constructing a sequence of these units, each building on the last, leading towards some goal state of the pupil. After each unit, testing may be used to decide whether to continue or engage in remedial study. Skinner described this "programmed learning" approach as a "technology for learning" rather than as an attempt at a theory of education.

1.1.2-Piaget.

A major difference in the approach to learning associated with Piaget is the idea of the learner as an active agent. The function of intellectual behaviour is regarded as adaption to the environment, and adaptive behaviour is regarded as a thing which is pleasurable in itself (there is supporting psychological evidence for this view e.g. [Holt 1965]).

Learning is seen as a process which changes the internal state of an organism by interaction with the world. This involves building new intellectual structures from old ones in order to achieve increasing complexity. Piaget proposes two techniques for modifying these structures;

1) Assimilation - absorbing new experiences into existing structures.

2) Accomodation - changing existing structures to deal with new situations.

Clearly, a real learning interaction involves both of these processes.

Piagetian models of learning have been widely accepted as a practical foundation of teaching methods. The exploration of the nature of education within this framework has received much study. A major contributor to this area is Bruner (e.g. [Bruner 1966]). An important focus of Piagetian approaches to learning is the provision of suitable learning environments to sstimulate the pupil.

1.1.3-Concept acquisition.

I wish to propose the following working definition of learning;

> " Learning is the process by which interaction with the environment brings about a (non-transient) change in the state of a system. This change may result in observable effects on the behaviour of the system, but this is not necessarily the case."

There are several difficulties with this definition. One is that the key phrase "non-transient" has been left undefined. Another problem is that this description does not allow certain processes which should be categorized as learning. A particular example is the insight into a subject which can be gained by simply restructuring knowledge internally. This is an instance of learning which does not require interaction with an environment. A further difficulty is that this definition permits negative as well as positive changes in the state of the learner - it allows someone to learn a worse way of doing something!

The type of learning on which this work will focus is Concept Acquisition. Within this area there is still a choice between behaviourist and cognitive models. The following is a somewhat behaviourist definition of "concept";

> " A concept is defined as a common response made to a category of experiences which have some property in common." [Borger 1966]

A more cognitive approach will be taken here. A concept will be regarded as an intellectual structure which imposes organization upon perceived events. These concepts achieve a more concise representation of events by representing them at a level of abstraction which has predictive power.

Learners will be regarded as active hypothesis-testers who are constantly generating hypotheses to explain their observations and testing them against the world. There is empirical evidence to support this view, and it has been shown that "..people pursue a definite information-seeking strategy which is guided by their current hypotheses." [Cohen 1983]

There has been considerable work on concept acquisition in the Machine Learning literature. It is this area that provides the best-formalized descriptions of concept learning. Some of this work will be discussed in more detail in chapter 11. We conclude this section with a definition of a concept taken from [Church 1963] , which is adequate to describe the term "concept" as it will be used in the remainder of the thesis;

> " Any given symbol can be attached to a set of objects. For any object there exists a rule to decide whether or not it belongs to that set. The decision rule is the concept represented by that name, and the set is the denotation of that name."

1.2-Theories of education.

Before discussing methods of educating people we must examine the difficult question "What constitutes an Educational activity?". The Oxford Concise English dictionary defines Education as follows;

> " Bringing up (of the young);systematic instruction;course of this...;development of character or mental powers;training (of animals)."

This definition shows the diverse nature of Education. What is the common theme of these processes which identifies them as Educational activities? Firstly we can see that they all imply the presence of some Agent other than the learner who is participating in the educational process. A person exploring her environment without assistance is not being "educated" in this sense. An Educational activity must certainly have links with learning. It is possible for learning to exist without education (as in the case of a young child examining its environment), but if learning does not exist then how could we define the goals of education? If we allow "learning" in its most general sense to be some process of adaption, then we may describe "Education" in general as acting on this process to affect the content (what is learnt) and the efficiency (how it is learnt). Within this definition, "teaching" may be considered as the activity which an intelligent agent (other than the learner) may make towards the achievement of some Educational goals. It is worth noting that this requires someone (teacher or pupil) to possess such goals.

In itself, this link to learning does not greatly reduce the space of possible educational activities, for learning, as we have seen in the previous section, appears to be a term covering a multitude of different activities. It seems that a description of "education" can provide us with standards which must be met by an educational activity, but cannot otherwise specify what constitutes such an activity. Peters [Peters 1966] summarises this as follows;

> " ...education is associated with learning... But no specific type of activity is required. A man can do it himself in solitary confinement, or acquire it by constant activity in a small group. He can be trained on his own by a tutor or inspired by lectures given to 500. In this respect "education" is rather like "reform". It picks out no particular activity or process. Rather it lays down criteria to which activities or processes must conform."

In order to specify a particular educational method it is necessary to provide a set of rules which describe what should happen in every conceivable situation which may arise during an Educational Interaction. It will not generally be possible to exhaustively enumerate every situation which may arise and specify an appropriate action for it. Instead, an attempt should be made to produce an educational theory that provides us with reasons for tutoring in a certain way so that it is possible to generate appropriate behaviour in a novel situation by referring to this theory. A further purpose of an educational theory is to allow comparison of differing methods of education.

Doyle [Doyle 1973] attempts to produce a more unified method of looking at education by providing a "theory schema" with variables which may be filled in to produce the Educational Theory of your choice! This reads;

> " Education is an activity in which X is fostering or seeking to foster in Y some disposition D by method M."

X and Y are agents of some kind. If X is omitted, we have a situation in which learning occurs without a teacher, and this can no longer be properly called education (although phrases like "life is my teacher" might suggest a more liberal interpretation). Y is clearly vital since the only changes which occur, occur in Y, and without change we have no process of learning or teaching. D is the crucial element because it has a major influence on M, and hence on most aspects of a particular theory. Doyle suggests that these "dispositions" could be particular states of mind, forms of thought, skills or attitudes. D and M together constitute the major components of an educational theory. Let us examine these components in more detail.

A reasonable description of the necessary units of an educational theory is given by O'Connor [O'Connor 1956] . He distinguishes three basic components;

1) a set of techniques for imparting knowledge, skills and attitudes.

2) a set of theories which purport to explain or justify these techniques.

3) a set of values or ideals embodied and expressed in the purposes for which knowledge, skills and attitudes are imparted and so directing the amounts and types of training that is given.

It is commonly suggested that there are two basic techniques which may be applied to the validation of educational theories; on the one hand are the methods of philosophy, and on the other are empirical techniques. In the list given above, component 3) is essentially a set of ethical and moral judgements. Such judgements can only be investigated by a philosophical approach. Components 1) and 2) however, involve testing hypotheses about methods for achieving given educational ideals and providing scientific theories to explain the efficacy of these methods. Problems of this type are amenable to empirical study by such disciplines as Psychology. Indeed, Bruner claims that these problems are central to Psychology, since "... it is psychology more than any other discipline that has the tools for exploring the limits of man's perfectability" [Bruner 1966] . It is not necessarily the case that philosophy can make no contribution to these areas since philosophical methods may well be appropriate to clarify the arguments; as Peters says [Peters 1973] ;

" We are, I think, only at the beginning of our understanding of what is a philosophical point about learning and what is an empirical point."

Peters examines the role which Educational Philosophy can play in deriving teaching methodology. He points out that the nature of philosophy is not such that it is ever capable of providing definitive answers to questions such as "What should I teach?". He does demonstrate its importance, though, as a tool to clarify practical methods and make explicit the assumptions which lie behind answers to such questions. Peters is not optimistic about this contribution of the discipline reaching practical decision-takers and affecting educational policy [Peters 1966] ;

"...educators...formulate principles in ignorance of the detailed discussion by philosophers of the fundamental asssumptions presupposed by these principles."

Peters finally proposes a goal of Educational Philosophy as being to;

"...indicate the sort of condition under which forms of conception could be intelligibly learnt. For with these, as with any other concepts, one has to postulate both conditions in which they could be applied and some aids to conceptualization, even if they take the form of "cognitive stimulation" rather than direct teaching. It is an empirical question, of course to determine the conditions which actually do have a marked influence." [Peters 1966]

Bruner indicates the complementary nature of these approaches when he points out that [Bruner 1966] ;

"... however able psychologists may be, it is not their function to decide upon Educational goals."

One point which Bruner makes is that a theory of instruction should be PRESCRIPTIVE. In this it deviates greatly from the DESCRIPTIVE nature of theories of learning and development.

Bruner identifies four major features which a prescriptive theory of instruction must satisfy;

1) A theory must specify those experiences which most effectively implant a predisposition to learning in the individual.

2) It should specify the ways in which a body of knowledge should be structured so that it can be most readily grasped by the learner. It is worth noting that there may be many possible structures for a particular body of knowledge and that the most appropriate structure is therefore a function both of the subject-matter and of the nature of a particular learner.

Bruner proposes several criteria for assessing a particular structure, including power to simplify information, power to generate new propositions, and power to increase the manipulability of a body of knowledge. This assessment will be discussed further below.

3) The theory should specify effective presentation sequences for the material.

4) It should specify the nature and pacing of rewards and punishment in the process of learning and teaching (Notice that this refers to intrinsic as well as extrinsic rewards).

In what follows we will reduce the idea of an educational theory to two components; a statement of a set of educational goals (which correspond to educational values to be maintained), and a specification of methods for achieving those goals (which includes both a statement of appropriate techniques and theories to support those techniques). The choice of goals lies in the domain of educational philosophy, while the methods of attaining these goals are open to empirical study.

1.3-The need for a new Theory of Education.

In this section an attempt will be made to provide reasons for advocating the development of a new theory to deal with Computers within an educational system. The section will also highlight some starting points from which such a theory could be developed. IMPART attempts to embody the approach to education which is described in this section. outlined.

The computer brings certain unique properties to an educational interaction. Various researchers (e.g. [Papert 1980]) have identified features such as the possibility of providing concrete representations of abstract objects, and have proposed applications for the computer as an educational tool based upon these features. There has been no attempt to relate these applications to the mainstream of educational thought, or to find a common framework within which to relate different approaches to computer based learning. These tasks would be fulfilled by research on the implications of computers for education theory.

One of the most important of the features offered by computer based teaching is the possibility of providing genuine one-to-one tuition (i.e. tuition adapted to the individual) for every pupil. It should be apparent that making full use of this feature is incompatible with many of the current educational theories. Since the time of Quintilian, educationalists have concentrated on single teachers dealing with groups of pupils. Many of the considerations in this situation are to do with controlling the group and producing a level of teaching acceptable to all members. To find discussion of individual tuition it is necessary to return to the work of Rousseau or Plato.

This concentration on one-to-one education revives many debates which were left unresolved when individual tuition became impracticable. In particular, computers bring the issue of freedom and constraint to the foreground. The power of the machine provides us with the opportunity to move further along paths of complete control of the population than has ever been possible previously. No existing theory has adequately dealt with these issues.

I would propose that this indicates the need for developing a radically new theory of education. Such a theory should attempt to describe the roles which may be fulfilled by a computer in an educational environment, and should endeavour to relate the possible activities of the computer to other activities in the broader educational setting. It may, for example, be the case that computers could be most appropriately applied as tools for imparting topic content to the pupil. Students would learn subjects (in the traditional sense) by interacting with the machine, which (due to the one-to-one nature of the interaction) would be able to support structured informal approaches more effectively than a teacher confronted with a whole class. This would free the teacher and pupils to spend time away from the computer involved in activites which foster personal development, acquisition of social skills etc.

I wish to suggest that producing a theory capable of handling the special requirements of computers may best be tackled by starting from the framework outlined by Jean-Jacques Rousseau [Rousseau 1762] . The following quote describes what Rousseau hoped for from the perfect educational environment. It is easy to imagine he is talking of a computer! In fact, he is referring to the environment described in "Robinson Crusoe";

> " Is there no way of correlating so many lessons scattered through so many books, no way of focussing them on some common object, easy to see, interesting to follow, and stimulating even to a child? Could we but discover a state in which all man's needs appear in such a way as to

appeal to the child's mind, a state in which the ways of providing for these needs are as easily developed, the simple and stirring portrayal of this state should form the earliest training of the child's imagination."

Briefly, Rousseau suggests that a child should have a single tutor who is with her from birth. They should engage in all activities together - there will not be one set of rules for the teacher and another for the student. The teacher may not directly control the pupil, but should manipulate her environment to ensure that appropriate learning experiences occur. Each new step must be integrated with the pupil's learning up to this point, and no idea is accepted unless it is understood and capable of being used.

The theory he expounds has a number of holes in it, and many of the ideas are inadequately worked out, but it is the most recent attempt to provide a complete educational theory dealing with all aspects of individual tuition. Archer [Archer 1916] offers the following summary of the standard criticisms of this theory;

> " The most frequent contemporary criticism to be made of the educational system of EMILE was probably that it was impracticable, and in many ways the criticism seems justified. Not only does the tutor have only one pupil..., but he must devote himself totally - and without payment! - to his pupil's education. The child must be kept in near isolation from society, and the tutor has to have absolute control over his whole environment. Though it is no doubt desirable for the child to learn only what he wants to, and then only when his interest has been aroused, is it really possible to rouse the ordinary child's interest as easily as Rousseau imagines? And however desirable it may be to acquire knowledge only by first-hand discovery, is it really feasible that an ordinary child would acquire enough in this way for his future needs, especially in the modern world?"

Many of these issues do not arise when the tutor is not human, but there are fundamental flaws in Rousseau's approach to education which require detailed study.

The first is that, while giving the appearance of freedom to the pupil, Rousseau actually constrains the educational experience very strongly. This is the reason that the tutor requires total control of the environment: nothing should happen without the tutors approval, but no overt interference should occur. Rousseau describes the task as follows;

> " Young teacher, I am setting before you a difficult task, the art of controlling without precepts, and doing everything without doing anything at all."
>
> " Do not undertake to bring up a child if you cannot guide him merely by the laws of what can or cannot be. The limits of the possible and the impossible are alike unknown to him, so they can be extended or contracted around him at your will." [Rousseau 1762]

The reason that such strong control is exerted is that Rousseau does not believe that his pupil is capable of the self-discipline necessary to follow a topic through alone, or of the foresight required to agree to subordinate one's own freedom to enhance the learning process. Most of Rousseau's work concentrated on an age when he considered a child far too young to understand the nature of educational goals. In consequence it is weak on ideas about learning as a joint responsibility of pupil and teacher, or about the possibility of the child voluntarily waiving her right to freedom in the interest of achieving those goals. Bennett [Bennett 1979] highlights this (still current) difficulty in the following way;

> " The question of whether pupils have sufficient maturity to choose a topic and carry it through lies at the heart of discovery learning

approaches."

Resolving this difficulty requires a better understanding of ways to foster self-discipline in a pupil and of circumstances under which individuals may reasonably be expected to voluntarily subordinate themselves to another. These areas both imply a model of the pupil which gives her more credit as a contoller of her own ends than Rousseau was prepared to give.

Another problematic issue is that Rousseau focusses upon the "natural" state of a human as the target to which he is guiding his pupil. This is very much Rousseau's own idea of what is "natural". While the state he describes can be seen to have many laudable aspects, there is no reasoning provided which gives this state any claim to primacy over any other state that could be thought up, and it certainly does not correspond to the "natural" state of human beings dissociated from a culture. The idea of being natural as opposed to "manufactured" is an enticing one which deserves further examination. An interesting perspective on this area appears in [Radcliffe Richards 1980] .

In the next two sections we will briefly summarize some educational goals and propose a method of teaching which is compatible with them.

1.3.1-Goals of an educational theory.

Autonomy.

Ultimately we wish the pupil to operate independently of the teacher. When this state is achieved, the only cues available to the student will be direct perceptions of the environment. In order to facilitate this the teacher must ensure that the student perceives relevant features of the environment. Further, the teacher must build upon these perceptions in guiding the pupil towards higher level concepts as methods of organising the environment. In this we are following Bruner [Bruner 1966]

> " Much of perception involves going beyond the information given through reliance on a model of the world of events that makes possible interpolation, extrapolation and prediction."

To achieve this, the teacher must avoid overwhelming the pupil with aid, and should provide as little assistance as is necessary at a particular time. Rousseau suggests;

> " Let there be no question of obedience for him or tyranny for you. Supply the strength he lacks just so far as is required for freedom, not for power, so that he may receive your services with a sort of shame, and look forward to the time when he may dispense with them and may achieve the honour of self-help." [Rousseau 1762]

we also find Bruner advocating such an approach;

> " The tutor must correct the learner in a fashion that eventually makes it possible for the learner to take over the corrective function himself." [Bruner 1966]

In particular, attempting to produce autonomous Artificial Intelligence programmers requires that we ensure our pupils are capable of dealing with a "typical" programming environment by the time their interaction with the system is finished. To achieve this end things are organised so that the pupil is always interacting with such an environment. In the early stages a teacher guides this interaction, but as the pupil develops the teacher will gradually withdraw, providing a smooth transition into the real world.

In order to maintain the consistency of the system which the pupil is trying to model, it is vitally important to foster a strong distinction between the teacher (whose nature is changing) and the programming environment (whose nature is fixed). The pupil is only expected to produce a powerful model of the latter. The teacher must encourage the pupil to interact directly with the environment to gain practical experience of the language being learnt. This also extends to leaving the pupil alone to tackle problems which are within her capabilities, using only the "built-in" feedback. Such problem-solving will aid the development of debugging skills.

Freedom.

In this context the term "freedom" is used to represent an issue distinct from that of autonomy. It is being used in the sense in which it is used in "free learning"; that is, to identify the issue of control within the Educational interaction. It is apparent that complete freedom is not necessarily good (e.g. we don't want our young children to be free to get run over), but in the restricted case where freedom implies a choice among "good" things, "...the enlargement of people's freedom tends to promote their interest because it provides more opportunities for the discovery of what is good" [Peters 1966] . We may now see the issue of freedom as one of only imposing restrictions on an individual if we can justify the imposition of those restrictions. The problem in an educational setting may be summarized as follows; "What constraints can justifiably be placed upon a learner during an Educational interaction?"

Two types of constraint must be distinguished. The first (type 1) is a limit on the ability of an individual with respect to the environment in which she finds herself. If a human flaps her arms up and down she will not be able to fly. Whether we can say that she is not "free" to fly may require further debate, but it is certainly apparent that the possible set of activities which she can carry out is constrained in this respect.

A second form of constraint (type 2) is that which is imposed by one agent upon another. If I wish to apply a bulldozer to 10 Downing Street I would meet a constraint (a policeman) which prevented me. This constraint exists because someone (probably the Prime Minister) does not want 10 Downing Street flattened. There is no intrinsic limit of nature which prevents me following this course of action.

In an educational interaction the distinction between these types of constraint is not so clear-cut. In the first case above the human cannot fly because of the restrictions of the "natural" world. An Educational environment is an "unnatural" world, and as such, while some constraints may appear to the pupil as limits of his environment, those limits were actually imposed by the designer of this artificial environment and are hence constraints of the second type.

Consider the example of teaching students to program using a syntax-directed editor. I (the teacher) have imposed this environment upon them because I consider the behaviour of syntactically incorrect programs to be irrelevant to teaching programming: in this sense it is a type 2 constraint (imposed by an agent). The pupils, on the other hand, will never see a syntactically incorrect program: such things do not exist in this environment, so this may be perceived as a constraint of the first type.

The issue becomes more complicated if we examine the relationship between the constraints imposed upon the pupil and the desires of the pupil. If the pupil is prevented from doing something which she will never want to do, then she will never meet the constraint and will be unaware of its existence. Is this a constraint any longer? I would suggest not. From this it follows that a teacher who guides her pupil so that the pupil only desires those things which are available to her is not constraining her in the same sense. Of course, since it is rarely possible to be certain that an individual would not hit the constraint if left to her own devices we must acknowledge the existence of some form of concealed constraint.

For Rousseau, the issue of freedom and control was the centre-point of his educational theory. It provides a source of hypotheses to which we will return later [Rousseau 1762] ;

> " The man is truly free who desires what he is able to perform, and does what he desires. This is my fundamental maxim. Apply it to childhood, and all the rules of education spring from it."

There is an imbalance between the free approach and the constrained approach to learning. If a pupil is free, then they are free to choose a more constrained learning style, whereas if the teacher always constrains the pupil there will be no scope for the pupil to move towards a less constrained approach. For this reason I claim that every teaching system should begin from a free learning perspective, and should be prepared to move to a more structured approach (it should also be capable of such a move).

Individuality.

Linked to discussions of autonomy and freedom we find the idea of individuality. Should we try and enforce some uniform approach to problems of a particular kind, or should we allow the idiosyncracies of the individual to develop, possibly even at the expense of efficiency? Rousseau gives primacy to the desires and beliefs of his pupil, and by linking new teaching to old experiences and discoveries produces a course of learning derived from the nature of his pupil. This is not "freedom" in the traditional sense, but the respect which it embodies for the desires of the pupil ensure a large component of freedom. Another advantage of this approach is that it builds upon the things which the pupil is most motivated to learn about, resulting in a more efficient interaction (see below). Claims may also be made that this immediate linking to personal experience increases the meaningfulness and memorability of the educational material. To quote Rousseau [Rousseau 1762] ;

> " The teacher's art consists in this: To turn the child's attention from trivial details and to guide his thoughts continually towards relations of importance which he will one day need to know, that he may judge rightly of good and evil in human society. The teacher must be able to adapt the conversation with which he amuses his pupil to the turn already given to his mind. A problem which another child would never heed will torment Emile for half a year."

Our choice is between encouraging the idiosyncratic programming behaviour of individuals, or teaching a uniform style to all our pupils. If the system was aimed at producing commercial software writers then we would choose the latter course. By ensuring a uniform style across our pupils we could produce software which is easy to maintain and debug (without reference to the original author).

In fact, the system is aimed specifically at postgraduate students learning Artificial Intelligence. In consequence we wish the pupils to develop a clear understanding of WHY certain language constructs behave in certain ways - using them without understanding is not adequate. To achieve this our system will permit a pupil to examine any area of LISP, the only restriction being that the pupil should understand fully that subset of LISP which they use.

This approach affects the way in which we assess the progress of a particular pupil. Part of that assessment depends on the degree to which the student has attained the domain-specific goals of the curriculum (e.g. knowing what a variable is), but a major consideration must be the level of understanding which the pupil has reached within that part of the domain which she has examined.

Authority.

If our pupil is merely taught to accept what the teacher offers, since the teacher is of special status and must therefore be right at all times, we have introduced two difficulties. On the one hand we have a stopping point in the process of understanding. It will be acceptable to say "X is true because the teacher said so", without looking for any further reasons. If our pupil once searches beyond what the teacher says, then this can no longer be used as a limit of understanding if we are to be consistent. The pupil must continue searching for reasons until she reaches some other form of limit (such as a relationship in the world). This searching beyond what the teacher gives is in some sense questioning the teacher's authority. The pupil will only accept the teacher's statements if they can be justified. Peters summarizes this point in the following way [Peters 1966] ;

> " Teaching involves further that, if we try to get the student to believe that such and such is the case, we try also to get him to grasp it for reasons that, within the limits of his capacity to grasp, are our reasons. Teaching, in this way, requires us to reveal our reasons to the student and, by so doing, to submit them to his evaluation and criticism."

Rousseau takes this point and sees it as implying that "authority by status" must be completely eliminated, and replaced by reason [Rousseau 1762] ;

> "Let him not be taught science, let him discover it. If ever you substitute authority for reason, he will cease to reason; he will be a mere plaything of other people's thoughts."

Our system will never make use of intrinsic authority. It has often been pointed out that any computer appearing to perform some mildly intelligent task is invested with a great deal of authority by a novice user. It is common to assume that " I am wrong because the machine doesn't make mistakes, so it must be right". It may be the case that the computer tutor must actively discourage such a perception of itself. I am not suggesting that the machine should make mistakes, but it should certainly leave room for uncertainty in its statements.

Any statement which the system makes to the pupil should also be justifiable to the pupil in terms which she can understand.

Pansophic teaching.

The problem of pansophic teaching was dealt with in detail by Comenius [Rusk 1967] . It is essentially a question about the general form of the content of an education. The contrast which Comenius drew was between pouring facts into an individual or attempting to instil a level of understanding such that facts could be discovered with ease. The former method provides expert knowledge in a particular area, while the latter encourages the development of general techniques which will contribute to many areas of the pupils' education. Comenius summarizes the difference thus;

> " Encyclopaedic teaching is neither practicable nor desirable; pansophic teaching is both. The one aims at making the learner an inexhaustible mine of information on every subject, the other would make him capable of wisdom in his regard for any subject and able to see any subject in relation to others and to general principles."

The emphasis of IMPART is on developing general skills of programming rather than memorizing particular details. At all times the tutor must try to fit experiences of the pupil into a broader context. The system should encourage the pupil to develop the skills necessary to think about problems at this more abstract level.

Motivation.

It is important to maintain the interest of the pupil in what is being learnt and to focus her attention on the matter in hand. Unless the pupil desires to learn, the task of education will be hopeless. This was pointed out by Locke [Locke 1706] ;

> "The greatest skill of the teacher is to get and keep the attention of his scholar; while he has that, he is sure to advance as fast as the learner's abilities will carry him."

Bruner recognises the importance of this issue and points out that curiosity alone is not sufficient since "... unbridled curiosity is little more than unlimited distractability". For this reason the curiosity needs to be constrained into certain channels without losing the "... energising lure of uncertainty made personal by one's efforts to control it". The desire for competence may meet difficulties if the pupil does not know exactly what is a reasonable goal to try and achieve. The tutor should foster a sense of accomplishment by identifying such goals. Preferably there should be well-defined tasks with a clear beginning and end, so that the feeling of "closure" when a task is complete may be invoked.

It seems self-evident that intrinsic motivations are preferable to extrinsic ones, since the latter will cease to exist when the teacher leaves or is no longer at a higher level of competence than the student. However, the pupil learning about a new domain has difficulty equating the desire to learn with goals within that domain. The tutor should provide extrinsic motivations while introducing the pupil to appropriate forms of goal within that domain. The pupil should gradually move towards the intrinsic motivations.

It is hoped that our pupils will be highly motivated to learn Lisp before they start using the tutoring system. However, as was discovered in earlier research [Elsom-Cook 83] this is not sufficient to carry them through the transition period when they are still unsure of the nature of reasonable Lisp goals.

To provide assistance during this period, the system should have some strategies for generating extrinsic motivations. The tutor must remove these motivations as the intrinsic motivations of using Lisp gain in strength. It seems necessary to provide some way of monitoring the usefulness of these techniques, otherwise the tutor will have no information available on which decisions about withdrawing the extrinsic motivation can be made.

1.3.2-Method of teaching.

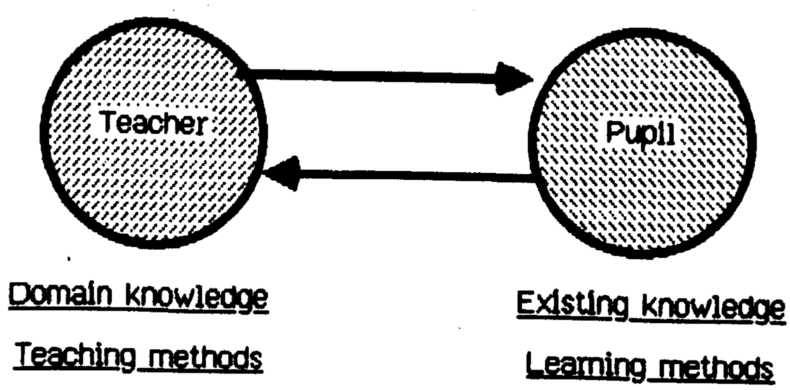


Figure 4 - Location of teaching knowledge.

In this section a particular teaching method will be outlined. Its consequences have not been fully explored, and the teaching method itself is not yet fully specified, but the tutoring system described in the second part of this thesis endeavours to use a style which is compatible with the method outlined here.

Consider a teacher and pupil engaged in an educational interaction. The teacher has knowledge of the domain to be taught, which she is able to apply in that domain. She also has knowledge of possible teaching methods and common pupil errors based upon past teaching experience. The pupil has certain existing knowledge structures to which new information will be related, and some preferred methods for learning.

These types of knowledge must be combined to produce an interaction in which the pupil gains well-integrated knowledge structures which can be applied to the domain. These will not necessarily be the same as structures in the teacher; partly because the teacher must bind them to different pre-existing knowledge, but also because the pupil may be better informed than the teacher in some aspects of the domain. A good teacher should help the pupil develop a model of the domain which is at least as powerful as her own.

The simplest approach to teaching which could be taken is for the teacher to externalize her representation of the domain as it stands. This leaves the pupil to impose some suitable order for learning, and to carry out the task of integrating this material.

It would be more helpful if the teacher could undertake part of this structuring task. The difficulty is that to structure the interaction well involves information about the domain (which is possessed by the teacher) and information about the internal state of the pupil (which is possessed by the pupil). Developing the best structure requires another level of interaction between teacher and pupil - essentially a negotiation about the educational style and content.

The teaching method which I would propose is as follows;

> The pupil interacts with an environment in which events related to the material to be learnt can occur. The pupil should be made aware of the sorts of assistance which the teacher can offer. The pupil uses the events which occur in the environment to learn about the environment. At

any time the pupil may turn to the teacher and request assistance.

The teacher observes the pupil and attempts to model the preferred learning styles of the pupil and the current state of the pupils knowledge. This information is combined with knowledge about the nature of the domain to enable the teacher to look ahead and attempt to predict what situations will arise in the (near) future. If the teacher predicts a situation arising in which the pupil will get "bogged down" in the learning process (examples of this would be exploring a blind alley, or consistently missing an important piece of information), then the teacher should intervene to guide the pupil away from this. If the teacher predicts situations in which the pupil is close to acquiring a new piece of knowledge about the domain, but not quite close enough to reach it alone, then the teacher should consider intervening to push the pupil towards that knowledge (there is a complex process of evaluation involved here). If the pupil appears to be losing interest in what is occuring, or experiencing a long period of unsucessful interaction, then the teacher should increase the pupils motivation - either by presenting an extrinsic reward or guiding the pupil to an activity with an attainable intrinsic reward. In other instances the teacher should do nothing unless explicitly requested to act by the pupil. Such a request indicates that the teacher has in some sense modelled the state of the pupil incorrectly, since no question should be unanticipated. Consequently, direct action from the pupil indicates that modifications to the internal model should be made.

Chapter 5.

Intelligent tutoring systems.

1.-Some descriptions of Intelligent teaching systems.

This section is essentially a description of several major systems which have been designed to approach the problem of computer use in education. The systems have been selected to provide examples of a wide range of perspectives on this area. The following section discussses some themes which link these systems together.

1.1-Scholar and Why.

SCHOLAR [Carbonell 1970] is a system which teaches facts about South American geography using a mixed-initiative dialogue. Domain knowledge is represented as a semantic net which links a fact to those around it with tags about relevance and appropriate timing for its presentation in a dialogue. Before designing this system, a study was made of the strategies used by teachers operating in a similar domain.

WHY [Stevens 1982] attempts to tutor knowledge about the causes of rainfall. It is provided with information about temporal and causal links between the facts which is used to generate some short-term structure in the interaction.

Collins claims that detailed domain-specific knowledge is "overwhelmingly central" to a teaching interaction. He says " The nature of the stored knowledge determines not only the content of a tutorial interaction, but also the goal structure that governs the tutor's selection of examples, questions and statements at different points in the dialogue, the types of misconceptions that students have and the way that tutors diagnose and correct these misconceptions based on student errors."

The system actually engages in a Socratic interaction, since it is claimed that this form of teaching is well suited to tutoring causal links between items. It has a set of 24 empirically determined tutoring heuristics which enable it to decide how to respond when it detects certain forms of error in a statement made by the pupil. These errors are in turn assigned to one of four categories. Having decided which errors need correcting, the system uses some ordering rules to decide upon a temporal sequence for tutoring them, and then generates its output.

Collins points out that the major practical difficulty with WHY is that it often detects and corrects the surface manifestation of an error, but does not detect the underlying cause.

The discourse mechanisms of WHY will be further discussed in chapter 12.

1.2-Trill.

The Rather Intelligent Little Lisper (TRILL) is a system designed to generate a didactic dialogue about basic functions and data structures in Lisp. It automatically generates sequences of examples, questions and explanatory text in an attempt to locate and correct errors in the student's knowledge about the language. A Socratic tutoring strategy is used, and the system progresses from correcting specific instances of error to revising general rules if the former technique does not work.

Knowledge about Lisp is stored as a semantic net whose nodes correspond to statements in Lisp and to concepts such as Function,Argument or data structure. These nodes are linked by relations such as Causal Property. An interaction corresponds to moving along a path through this net, presenting any canned questions which are encountered en route. Properties can be inferred from the superset to which items belong. For example;

Selector

.

CDR

.

Remove first element

.

Identify first element

could be used to solve a problem of extracting an element from a list by first generating an interaction about selector functions in general and then CDR in particular. If the action of CDR is not clear, the system may revert to testing whether the pupil can identify S-expressions within a list. The net is described as " a formalization of intuitions on the mistakes students make and on the procedures for diagnosing and correcting the misconceptions underlying these mistakes." [Cerri 1983]

The student model of the system consists of tags attached to items in the net. The system presents questions to test whether the student has a correct grasp of a piece of knowledge, and will tag text which it has presented, questions which have been correctly answered and practical knowledge which has been verified accordingly.

1.3-West.

WEST is designed to coach a player using a simple game called "How the west was won" (a more complicated version of snakes and ladders). The game involves use of basic mathematical skills together with some strategy particular to this game environment. The goal of the system is to develop the pupil's ability in these areas without detracting from the enjoyment of the game.

The environment is one in which direct teaching strategies (such as Socratic tutoring) are inapplicable. WEST tackles the problem from the perspective of guided discovery learning. This involves providing a "learning environment" which is augmented by a system to provide tutorial guidance for errors whose symptoms are beyond the student's ability to recognise or correct. The idea of a bug or misconception is central to this approach. A "constructive" bug is a difficulty which the student is able to learn from during the process of correcting it. The environment should encourage constructive bugs and discourage non-constructive ones, but since there will necessarily be non-constructive bugs, the teacher must have a goal of turning such bugs into constructive ones. Examples of non-constructive bugs are problems which can only be solved by techniques that are a long way beyond the current skill level of the pupil, or bugs which are not perceived.

It is not claimed that Guided discovery learning is the only way to learn, but that it is a technique worthy of further investigation. It is pointed out that "Many human tutors interrupt far too often, generally through lack of time or patience." [Burton 1982]

WEST has a model of an expert player which generates optimal moves in each game situation. The system assesses a player's move by comparing the outcome of that move with the outcome of the optimal one. If the move is sub-optimal, the system attempts to resolve both moves into a sequence of underlying steps and identify any discrepancies in the skills or concepts which are used. If the system identifies a discrepancy and decides to tutor it, then a talker associated with this issue is activated.

ISSUES represent the basic domain knowledge in the system. Each issue represents some skill which the system monitors. It has a weighting which is derived from the amount of evidence which the system has collected for the existence of that skill. An ISSUE embodies some of the glass-box model of problem solving in the domain. It is able to recognise an application of itself in a problem solution, and has some canned text associated with it to enable it to talk about itself.

Brown points out that it is not necessary for a system to provide a problem solver which actually solves problems in the same way as a human (a glass box expert). So long as some problem solver exists (a black box expert), all that is necessary is to provide the facility for generating glass-box reasoning for a given problem solution. This permits some contribution to be made by a tutor in domains in which glass-box knowledge is incomplete.

The problem of assessing the pupil is difficult in a gaming environment, since diagnostic techniques must not interfere with the game. WEST builds its student model by observation of the actions which the student uses. There are several problems with this, for example; "With the expert it is not possible to determine whether the student is weak in some skill, or whether the skill has not been used because the need for it has arisen infrequently in the students experience." Several problems arise with WEST because it has not tackled the way in which the student model must change with time. A skill which the student has just met for the first time will appear the same as a skill which the student has known for a long time, but is weak in using.

Burton identifies three major sources of noise in the system. One is that blame is uniformly assigned to all issues involved in a bad problem solution, so it is difficult to uniquely identify a problem area. A second problem is that the reasoning generated by the differential model is not necessarily that of the pupil, and the inconsistencies may confuse the modelling system into tutoring the wrong issue. A third cause of noise is that there is no model of deterioration of the student's performance due to fatigue or boredom.

1.4-Wusor.

Like WEST, this system is designed to coach a player in a game environment. In this case, the game is Wumpus which is an exploration game that exercises logical skills associated with making inferences from evidence presented in order to choose a move. The game is simple, and a complete model of all possible strategies in this closed environment can be easily generated.

The basic model of the domain knowledge is a complete enumeration of rules which can be used and the rule-variants which represent bugs but may be used by a learner. Goldstein is opposed to systems which simply model a learner as a subset of these rules. Instead, WUSOR attempts to capture some information about the relationships between rules and the ways in which they are learnt. The rules are structured by linking them with one of five relations;

GENERALISATION- R' is a generalisation of R if R' is obtained from R by quantifying over some constant.

SPECIALISATION is the inverse of generalisation.

ANALOGY- R' is analogous to R if there exists a mapping from the constants of R to the constants of R'.

DEVIATION - R' is a deviation of R if R' has the same purpose as R but fails to fulfill it in some circumstances.

CORRECTION is the inverse of deviation.

Note that the last two links will necessarily lead to buggy rules. This structure of interlinked rules is called a Genetic graph.

Learning is viewed as the process of creating new rules from those already existing by following these links. The student model consists of tags showing which rules have been learnt and which links have been followed. The system can then produce teaching statements designed to take students to a nearby rule via a type of link which they like using.

The structure of the graph provides a possible mechanism for recognising unintentional actions by the student(slips). A slip may produce a result which is accidentally worse than expected, or accidentally better than expected. In either case, it must correspond to an item in the Genetic Graph which is distant from the current boundaries of the pupil's skill level.

1.5-Buggy.

BUGGY is not strictly an intelligent TEACHING system, since it's purpose is to serve as a tool for diagnosing difficulties rather than remediation. It has been applied to tutoring in the system DEBUGGY which was used to help teachers learn to identify the bugs of pupils.

The domain in which BUGGY operates is that of high school subtraction problems. The reason for choosing this domain is that subtraction "... is a virtually meaningless procedure. Most elementary school students have only a dim conception of the underlying semantics of subtraction...This isolation...allows me to study a skill formally without bringing in a worlds worth of associations." [VanLehn 1981] . Prior to the BUGGY system, some educational researchers had done work on analysis of the bugs underlying children's arithmetic errors, and books to tutor teachers in detecting bugs in such skills had been produced [Ashlock 1976] .

BUGGY operates by attempting to build a model of the pupil which accurately predicts her response to test problems with reference to the features of those problems. The system does this by selecting rules for solving problems from a collection of correct and "buggy" rules which it knows about. It varies the choice of rules to provide the closest possible fit to the results which a student achieved on a pencil and paper test. Data from BUGGY has enabled the detection of over 300 types of bug which have been used to augment the rule-base. IDEBUGGY differs from the basic system in that it interactively presents subtraction problems to the pupil, so it is able to specify problems which are rich in the bugs which it believes are present, thus enabling it to verify its hypotheses more quickly and accurately.

The simple model of bugs does not explain all the observed errors. It was found to be necessary to postulate two extensions to the theory. One is the slip, which is a surface error which does not appear to be a manifestation of some underlying problem. BUGGY identifies these by assuming that they have virtually no temporal stability. The second extension is the concept of a repair.

The assumption of repair theory is that when a person comes up against a difficulty in solving a problem, they are unlikely to give up. Rather, they will attempt to use the knowledge that they have to find a way to circumvent the difficulty. These heuristics for patching up an error are known as REPAIRS. Repair theory states that a surface manifestation of an error is due to the attempts of a problem solver to repair a problem which has arisen due to one (or more) bugs in their knowledge. The following example illustrates this for a simple subtraction problem;

```
23 -        23 -
19          19
--          --
16          10
```

Both the above problems are manifestations of the SAME bug - a missing rule about borrowing when the top digit is smaller than the bottom one. The answers differ because a different repair has been applied in each case. In the first sum, the reasoning would be something like "3-9 can't be done, so do

9-3=6", while in the second case it would be "3-9 can't be done, but 9 is bigger than 3 so it must be 0".

Empirical results show that while a bug/slip model explains about 60% of children's subtraction errors, adding repairs allows the model to explain almost 75% of errors [VanLehn 1981] . Most of the errors which are not resolved seem to involve temporal instability of bugs. An attempt to deal with this by permitting TINKERING (using different repairs for the same bug during a test) and BUG MIGRATION (changing the repair used on a bug between tests) has been made, but no good model of the stability of bugs exists.

1.6-Guidon.

GUIDON is a teaching system which was designed to operate over a knowledge base already set up for the expert system MYCIN. MYCIN attempts to diagnose bacterial infections by reasoning from information about the symptoms of a patient. It can ask clarifying questions and suggest methods for examining the patient. The MYCIN system has a rule base containing approximately 450 rules, of which about 20% are typically involved in the solution of a given problem.

The problem domain is one in which the heuristics for diagnosis which the student is being taught are not derivable from a simple model of the problem. This lack of causal linking between rules simplifies the task of tutoring. Some abstractions such as rule-groups are discussed which attempt to capture some hierarchical structuring of the rules (presumably reflecting the structuring in an expert problem solver), but since such information is not currently present in the MYCIN database a closer approach to modelling the organization of the experts knowledge involves rewriting all the rules.

GUIDON teaches by presenting the pupil with a case history of a patient and allowing her to ask questions and to state hypotheses about the case. The system compares the questions and deductions with those made by MYCIN and tutors on the basis of these differences, hence it is a form of differential modelling.

The major difference between this and other tutoring systems is that GUIDON concentrates on generating a structured interaction. The dialogue is mixed initiative, and goal directed. The "knowledge of communication" within the system includes a model of the students knowledge, a list of the tutors intentions and a list of the student's intentions. This knowledge is represented as a subset model of the student together with a "focus record" and a "case syllabus". The focus record keeps track of factors in which the student has recently shown interest. The case syllabus is a list of those topics which are vital to the current problem. This could theoretically be generated automatically from MYCIN'S solution, but is actually produced by hand before a session. The actual course of topics discussed is determined by combining these three items with a general weighting for importance assigned to each topic.

The discourse mechanisms of Guidon will be further discussed in chapter 12.

1.7-Logo.

1.7.1-The basic environment.

The Logo system produced at MIT is a programming language environment designed to be used as a teaching aid for children. The general spirit is to encourage exploration of some command subsets (MICROWORLDS). The best known microworlds are a world of "turtles" which allow study of various geometric systems [Abelson 1980] and microworlds for learning about music [Bamberger 1976] . LOGO is in use at many centres throughout the world, with varying degrees of faithfulness to the original conception upon which it was based.

The educational philosophy which is implicit in the original statements of the LOGO approach has much in common with the child-centred education movement. The impression given by much of the literature is that it is only necessary to sit a child in front of a LOGO computer and the child will guide her own learning experiences. A teacher is expected to be accessible, although her role should be more like that of a companion in exploration. Papert offers the following overview of this role;

> " The LOGO teacher will answer questions, provide help if asked, and sometimes sit down next to a student and say 'Let me show you something'". [Papert 1980]

No further details of the teacher's goals are given.

Papert claims that the procedural nature of LOGO makes the learning process easier since we are used to interacting with the réal world, and hence procedural knowledge is a major part of our everyday activities. LOGO is seen as a tool to give the child greater access to her own thought processes. It encourages the student to think about thinking by providing a concrete external form for her beliefs in the shape of a program. In the course of developing and debugging a program, Papert claims that the pupil will develop and debug her own beliefs. This is not dissimilar to the role of a computer in Artificial Intelligence research.

In practice, most LOGO centres structure the experience which the child will have by setting problems, providing worksheets etc. [Howe 1982] . The most successful interactions are those in which one adult and one child operate a computer together. Caution should be exercised in deciding how much of this learning can be attributed to the LOGO environment, and how much is due to individual tuition. As an example, let us consider a case study of LOGO use at MIT. The LOGO laboratories at MIT have maintained a strong allegiance to the principle of learning experiences driven by the pupil. However, we find that the only case study available from MIT [Solomon 1976] describes a lesson very much at odds with this approach.

The study deals with a child who is having a 40 minute session with a LOGO system, after spending 4 months using LOGO then 6 months without it. During the whole session the child receives one-to-one tuition from Cynthia Solomon and in addition Seymour Papert is in the background, though he only intervenes at one point. Various points in the interaction can be identified at which there is scope for the child to take the initiative. In fact the child never does, and every idea originates from Cynthia Solomon. Solomon states "From the sessions with Lin, another first grader I had worked with quite extensively, I developed techniques and aids which have helped older children get into turtle work, and subprocedurisation, debugging, anthropomorphising." Solomon appears to have a large number of projects and materials prepared in advance of a session. She says "...I was always ready to intervene in case the situation became unresolvable ..." This intervention is often of a minimal form such as simply reading out things which the child has already written down.

1.7.2-Spade.

Spade provides a heavily structured programming environment within the LOGO system. It is intended to teach students to plan a program by writing a detailed specification. This specification is in a language devised for the automatic turtle-program debugger MYCROFT [Goldstein 1980] .

The system represents the planning process as a sequence of steps chosen from categories such as DECOMPOSE_PROBLEM or REPEAT. This is represented as a context-free grammar, and deriving a valid plan constitutes making a set of choices from this grammar. It is claimed that this automatically leads to a taxonomy of bugs in the system. For example, a syntax bug violates the grammar, a semantic bug omits a constraint of the problem (e.g. includes an unnecessary optional component of the grammar), a slip is anything which does not reflect a conceptual error in the plan etc.

The system operates by prompting the student to perform a particular type of planning at each node in the tree;

> "At each step SPADE-0 chooses an appropriate next step to pursue;but there is no requirement to accept its choice...Although the system always encourages obeying the model, features are provided to allow violating it."

This choice of an appropriate next step corresponds to a model of the "ideal" student. SPADE does not attempt any modelling of individual users, but regards pupils as variations on the ideal model.

The system has several "modes" which correspond to different stages of the process of generating a program. Refinement mode allows development of the plan. There are also modes to try out a program, locate a known bug and repair it by editing plan or LOGO code. The system switches between these automatically at appropriate points in the interaction.

2.-Themes in tutoring system design.

In order to design an Artificial Intelligence based system for tutoring it is important to consider earlier systems which have been produced to perform similar tasks. This section attempts to identify some themes which should be considered when attempting to make an evaluation of Intelligent Teaching Systems. It is hoped that these themes are sufficient to provide a framework within which comparative examination of teaching systems may take place. Some new terminology is introduced. Throughout this section two questions should be borne in mind. The first is "What requirements MUST a teaching system satisfy?" and the second is "What can a computer bring to the task of teaching other than the facilities provided by a human?".

2.1-Philosophy of approach.

In the sense that a human teacher can be described as applying a particular philosophy of education, it should be possible to identify the philosophy of any given tutoring system. Selecting such a philosophy should be one of the design decisions taken in developing the system. Since these teaching systems operate in a one-to-one environment which is unusual for human teachers, it does not immediately follow that a system has an adequate philosophy if it embodies that of its designer. In practice, most systems teach from a set of adhoc rules generated by a single individual. A goal of tutoring system design must be to explicitly embody the educational philosophy within the system so that it may be changed. A reasonable intermediate stage is to specify the philosophy on paper in advance and build the system in accordance with that philosophy.

The most common philosophy applied by existing systems is that of Socratic tutoring. There are practical reasons for this. In the Socratic method the teacher is very much in control of the interaction; the options open to the pupil at any given point are restricted. This makes the task of understanding the students actions much simpler. SCHOLAR, WHY, TRILL and GUIDON are all examples of this approach.

Another possible philosophy is that of Discovery learning. In the ultimate form, "free" discovery is epitomised by the LOGO approach. The pupil is set in front of a computer with no preset task. The difficulties of this method have been discussed above. It may be observed that this approach to LOGO use echoes the child-centred movement in Education and the progressive education movement. The similarity extends to include the unresolved problems associated with these approaches. More interesting application of Discovery

Learning is in the Guided discovery learning approach of systems such as WEST and WUSOR. In these systems the pupil is playing a game, but a tutoring system is observing the interaction in search of points at which Educational Input may usefully be interjected.

2.2-Communication problem.

A real human teacher makes use of many forms of communication during the course of an interaction. Non-verbal cues such as facial expression and features of spoken language such as pauses and intonation greatly enhance the amount of information which the teacher can gain from his pupil. By comparison, a computer has a very restricted range of information about the pupil which it can access, and it is unlikely that any extra information sources will become available to computers in the near future. Imagine trying to teach someone when you are unable to see visual cues or hear vocal ones. The loss of these information sources makes the teachers task much more difficult. This is the problem faced by a computer-based teacher: it has a narrow channel through which information must pass. Teaching systems must search for ways to extend the effective width of this input channel. Given that techniques such as speech recognition are too little developed to make actual broadening of the channel possible, there are two alternative approaches to the problem. The options are either to explicitly obtain all the information which is required through a channel of narrow bandwidth (which is a long and tedious process) or to rely on inference techniques to make assumptions about likely situations. A human teacher also uses inference in this way, but the paucity of input to the machine means its inference techniques must be more powerful. A clear example of the use of wider information input occured when the BUGGY system had its performance on detecting errors in children's subtraction compared against human teachers. In almost all the cases where teachers correctly diagnosed problems which BUGGY missed, the diagnosis was achieved by reference to the "scratch marks" made by pupils during the calculation. If this information was made available to BUGGY, we could say that the bandwidth of the human-computer interaction had been broadened.

From this description it is clear that the problem of increasing the richness of communication with the system is very much tied to considerations of the sort of information which is needed by the system. Assessment of the "value" of the input information can only be made with reference to the task for which that information will be used. This clearly indicates the necessity for using knowledge of the domain being tutored and knowledge about the pupil (the user-model) when attempting to expand the interaction.

At the simplest level, a computer just has access to a sequence of key-presses. These cannot be regarded as a form of communication between pupil and machine until the machine is capable of assigning meaning to these events at some level. The most primitive meaning is that assigned by a computer operating system such as UNIX. It has enough "knowledge" to recognise groups of characters which correspond to individual commands and to find the program corresponding to a particular command and channel the interaction to it. Determining higher levels of meaning involves going beyond the surface form of an event and trying to interpret the purpose behind it. This introduces the assumption that the computer user has "goals" which she is attempting to satisfy through the interaction. Since there is no need to assume that a single goal is created and satisfied for each event in the interaction, this brings with it the requirement for a model of the set of current goals of the user at any given time, and this in turn requires a representation for possible goals and methods of satisfying them in the domain with which the interaction is concerned.

Given that a computer has limited access to the information about the pupil which it requires to teach, there are several ways to get round the problem. All require that some knowledge of the domain being tutored is used to guide (or process results of) the interaction. The simplest method is simply to ask the pupil if she knows something. Socratic teaching systems, such as SCHOLAR or WHY, typically engage in such interaction. Since it is not always the case that a pupil knows that she knows something, and some skills may be present at a variety of levels, we find systems such as WUSOR do not "believe" what they are told, but regard direct statements by the pupil as evidence towards something. A more complex method of eliciting information is to provide a teaching system with test-questions (or the facility to generate such questions) whose outcomes give information about various aspects of the knowledge which is being tutored. As these questions become more complex, successively more powerful inference mechanisms must be used to interpret the implications which they have for the systems beliefs about the pupil. IDEBUGGY may be considered as an example of this technique, if we regard a single subtraction problem as a question and the whole process of bug analysis as the inference mechanism linking that question to the systems beliefs about the pupil.

In systems which are tutoring a skill, some scope must be given to the pupil to try out that skill. Monitoring the attempted use of the skill is a way for the system to collect extra information without interfering with the pupil by making this collection process explicit. SOPHIE, for example, monitors a pupil attempting to fault-find in an electronic circuit. It attempts to use the information to guide its tutorial strategy. The system often enters a direct interaction with the pupil to clarify its observations of the fault-finding process. Since the system does not, in general, control the behaviour which it is monitoring, the inference techniques necessary to derive information from such a process must be even more powerful than those associated with asking complex questions. This is similar to the difference in difficulty between learning from examples and learning by observation. In its most extreme form this can be seen in those systems which Burton [Burton 1982] classified as "coaching" systems. WEST and WUSOR are examples of such systems. The fact that they operate in game environments means that direct intervention must be restricted to the level at which it does not affect the enjoyment of the game.

The bandwidth problem which affects communication from human to computer does not have a counterpart in communication from computer to pupil. A computer has a wide range of output forms available to it such as high level graphics, speech (typed natural language) and special teaching devices (e.g. Turtle, Slide projector). The tutoring system can integrate all these things (which a human teacher must regard as separate media) into a single coherent interaction.

In summary, then, effective communication requires the computer to go beyond the information explicitly obtained from the pupil. In order to do this the system requires information about the domain which it is tutoring and about the current state of the pupil. This should include a model of the Goals of the pupil.

2.3-Choosing a problem domain.

The choice of problem domain has an effect on almost all aspects of the system. It is useful to investigate teaching by considering specialized domains, but the consequences of the domain choice for the rest of the system must be examined. In particular, it is important to consider whether some aspects of the domain mean that the system uses techniques which cannot be generalized to other tutoring tasks.

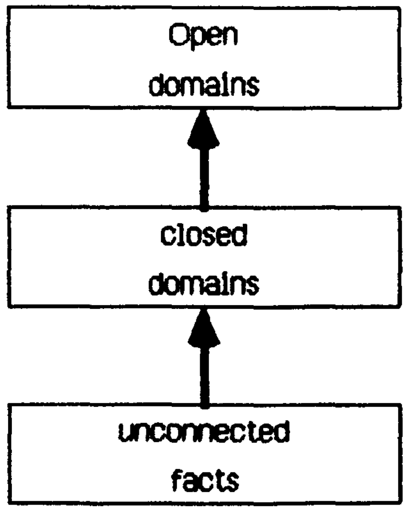


Figure 5 - Types of tutoring domain.

The simplest domain is probably that comprising a set of unconnected facts (e.g. Derby winners). Since the system must know all the facts, and since there is no underlying linking mechanism which must be tutored, the pupil cannot bring any information to bear on the problem which is outside the knowledge of the system. Previous knowledge of the pupil can have no effect on the task.

It is clearly the case that few domains support this idealized form of knowledge acquisition. Since many domains come close to this ideal, there exists a number of systems which tutor as though their subject area was of this type. An example of such a system is SCHOLAR, which attempts to tutor facts about South American Geography which cannot be derived from more general rules. Perhaps a more surprising example is GUIDON. In this system the domain consists of "rules of thumb" used by medical experts. The experts cannot normally provide reasoned links between these heuristics, so in some sense these must also be tutored as unconnected facts.

A more complex domain is the "closed environment" in which all possible correct methods of relating facts can be enumerated. By providing a restricted set of possible skills, such an environment simplifies the problem of user modelling and ensures that the system can behave as though it has complete knowledge. WHY is an example of such a system. The introduction of reasoning steps in the learning task complicates the issue by allowing previous experience of the pupil to affect the learning process. However, the closure of the domain ensures that no matter what course is chosen it cannot pass outside the knowledge of the system. Such domains permit the design of an "expert" who knows everything, and may allow the complete set of possible "experts" for the domain to be generated. Almost all Tutoring systems to date operate in a closed environment.

An open domain is one in which there are an infinite number of possible problems. In such a domain, the computer cannot assume that it has complete knowledge of the problem area and methods of problem solution. This is more like the problem which faces a real teacher. The system must possess methods of assessing, tutoring and modelling the pupil which can deal with situations which are new to it. In short the pupil must learn A right way to solve a problem rather than THE right way. This has implications for user modelling and assessment of pupil progress which will be discussed below. Many systems deal with domains in which a number of possible correct behaviours exist by modelling only one of these behaviours. Pupils must use the mechanisms which the tutor knows, even though these may not be the most appropriate mechanisms for the pupil and domain.

It should be noted that real tutoring tasks may often be usefully divided into more than one domain, with the tutor attempting to teach all the domains in parallel. A common example of this is domains which require certain levels of meta-knowledge in order to use them. It is often the case that skills required for the application of information may be regarded as a separate domain from acquisition of that information.

Having established the domain in which the system is to operate we must also consider how to keep the pupil within that domain. If the tutor is asked a question outside its area of specialization it should acknowledge its limitations. From this it is apparent that effort must be expended on providing the system with methods for deciding upon its own limits.

In summary, the choice of domain must be examined to see what effect it has on the general applicability of system design. In particular, the distinction between closed and open domains must be made.

2.4-Representing domain knowledge.

Having chosen our domain, we must decide how to best represent it for the task of tutoring. This knowledge is static, yet there are generally relationships which hold between the components that must be modelled.

SCHOLAR, WHY and TRILL all use a semantic net to represent concepts to be taught. These topics are organised into a hierarchical structure which is primarily based upon levels of generality. A particular topic may be a sub-component of a general issue and may itself possess other topics as sub-components.

The Genetic Graph representation introduced by Goldstein imposes a rather different type of ordering. In this system domain knowledge is represented as little islands of knowledge. These knowledge units are connected by links which correspond to different learning methods, such as generalization, analogy etc. Goldstein uses this mechanism as a first attempt at modelling the processes by which a learner may negotiate a body of information.

It is also worth mentioning GUIDON, since this system attempted to tutor using an expert system (MYCIN) as its domain-knowledge source. This approach was not found to be adequate. More recent research is attempting to rewrite the expert system in order to include knowledge which is better able to support tutoring interactions. It remains to be seen whether there are fundamental representation problems with an expert-system based approach.

2.5-Modelling the pupil.

In order to tutor effectively, a system requires knowledge about the current level of ability of the pupil. This information can then be applied to selecting an appropriate item to tutor and a style for teaching it which will relate it to those areas in which the student is currently interested. In practice, the form which this "user model" takes will be constrained by the range of techniques available for eliciting information about the pupil and by the application to which this knowledge will be put. The former limitation was discussed above; In this section we will consider the effect of the latter constraint. The type of model which is possible depends on the type of domain in which the tutor is operating; this section reflects the ordering of domains.

The simple domain of independant facts can be tutored with a fairly basic model of the learner. Since each fact is either present or absent, and no hierarchical organisation can be imposed, it is just necessary to associate a tag with each fact indicating its presence or absence (e.g. SCHOLAR, TRILL). The problem may be slightly more complex because learners may "forget" something which they have learnt, or make a "slip" (a careless error) in what they are doing. For this reason the mechanism associated with these user-modelling tags should be able to remove tags and note the temporary instability of tags as well as adding them to items which the pupil has learnt. The unprincipled "slips" may be described as "noise" in the interaction which the user model must overcome.

In an appropriate environment, this technique can be very powerful. EPAM [Feigenbaum 1963] used a discrimination net to memorize nonsense syllables. This simple mechanism demonstrated much of the behaviour shown by humans when confronted with this task based upon meaningless elements.

Let us now consider a closed environment in which there are relationships between basic "facts" and every possible solution method can be enumerated. It is possible to define a set of "experts" who could operate correctly in the domain. The goal of the system is to make the pupil an expert in the domain, and since the teacher knows all the possible experts this is equivalent to making the pupil into one of these experts. The idea of glass-box and black-box experts [DuBoulay 1980] which represent the division between "psychologically reasonable" and "psychologically unreasonable" forms of expert may now provide a dimension for categorizing the usefulness of particular experts as models which may be directly tutored to the user.

Given that the particular expert which we are attempting to tutor corresponds to a set of skills E, that our pupil has a set of skills P, and that there exists a set of skills B which may be acquired by our pupil but which have no counterparts in our expert in the domain, we may represent our pupil as an OVERLAY model. There are two types of overlay model, subset models and perturbation models;

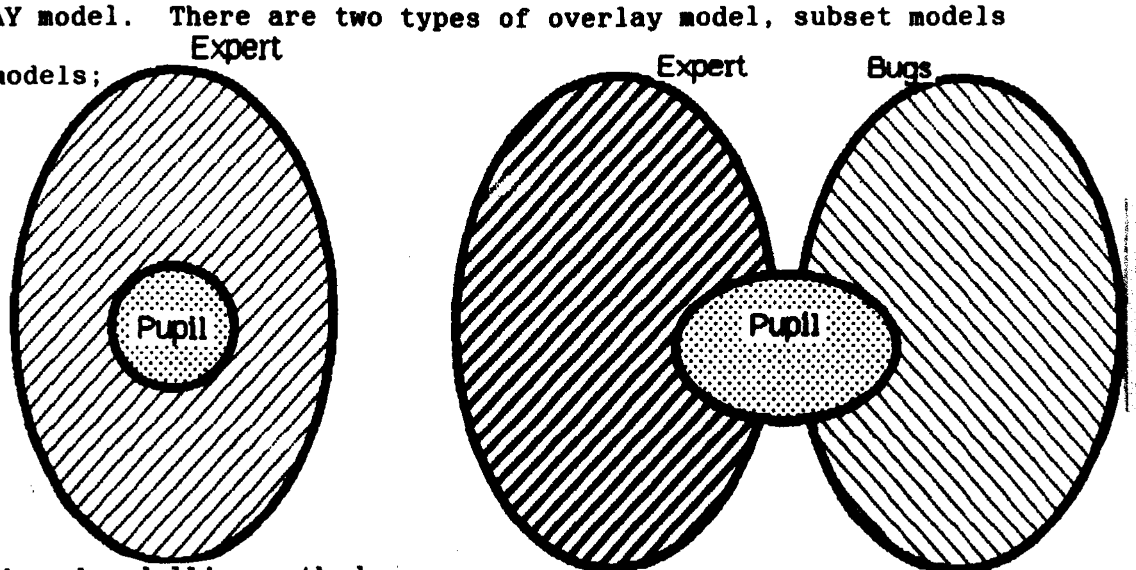


Figure 6 - Expert-based modelling methods.

The subset model means that for each skill which exists in the expert, the pupil either has that skill or she doesn't. The teacher will try to expand the pupils skills until they are equivalent to those of the expert.

In a perturbation model there may be skills possessed by the pupil which do not have counterparts in the expert. These skills are faulty or "buggy" and will not lead to correct behaviour. A teacher with a perturbation model must attempt to eliminate all "buggy" rules which the pupil possesses and expand the set of skills common to expert and pupil.

Systems which rely on perturbation modelling (e.g. Buggy,LMS) ultimately reduce the model to a set of primitive subskills which are either present or absent - a buggy rule corresponds to a set of subskills which differs in one or more components from the set associated with the correct rule. It is worth noting that bugs, like the domain knowledge itself, may also correspond to a closed or open domain. In essence a bug-based system is an expert-system for mistakes: it suffers from the same limitations as expert-based domain knowledge. In particular, it does not know how to handle classes of errors which it hasn't been explicitly told about, and will persistently try to coerce the student into the categories which it has available.

Another term often used in the literature is "differential model". This means that the pupil is modelled in terms of the difference between pupil and expert. This is essentially equivalent to an overlay model, but the focus is on reducing the size of a set of "differences" between pupil and expert rather than increasing the set of expert skills. This variant forms a valuable assessment technique.

Some systems (WEST) attempt to tutor by achieving equivalence between the pupil and a set of partial expert models which do not cover the whole domain. These "local experts" have the same fundamental nature as complete domain experts.

WHY operates on subset modelling. It has a set of causal links between facts about rainfall which correspond to a particular theory of rainfall. A numerical value is associated with each link to represent the amount of evidence which the system has that that link has been acquired. Not only does WHY not have a representation of possible incorrect rules which a pupil might use to model rainfall, it only represents those rules corresponding to one particular expert in the domain.

At this point it is worth noting that tutoring, diagnosis and assessment are different functions of the pupil model. Expert based systems provide a useful set of criteria for assessment since they constitute a benchmark against which students can be compared. This comparison cannot validly be extended to the internal workings of the pupil and expert. In trying to use an expert for tutoring in this way systems overreach the usefulness of the technique.

The techniques applied so far may prove valid in a closed domain, but once we attempt to tutor an open domain the problem of user-modelling becomes vastly more difficult.

In general, it is not possible to enumerate all the conceivable problems and solutions within the domain, so an individual's domain knowledge must exist in the form of some generative model of the domain which predicts its behaviour under various circumstances. It will not generally be the case that one "correct" model of the domain exists. Each model must be assessed in terms of its own virtues and vices. Issues such as input-output behaviour, simplicity, "prettiness" etc. must be considered. It is likely that different individuals will have different internal structures for their models, since they are linked to different pre-existing structures in the individual. For this reason a teacher should be prepared to accept any model of the domain (not just her own) provided it satisfies assessment criteria such as predictive power. An example of such a situation as this would be if a tutor knows only Newtonian physics and is attempting to teach simple mechanics problems. If a pupil consistently produces answers equivalent to those of the teacher by using different methods (e.g. the laws of relativistic physics) then the teacher must accept the students' model of the domain. In this case, it is found that the student has a more powerful model of mechanics than that of the teacher since it will predict correctly when Newtonian physics fails.

It is useful to describe open domains as ones in which the pupil must acquire certain concepts if she is to understand the domain. Concepts may be acquired in closed domains, but they are necessary in open domains. A concept is a means of organizing information.

In order to model the state of the pupil in an open domain, it is necessary to model the concepts which the pupil has acquired. If the intention is to find ways to educate the pupil (i.e. extend her model of the domain), then it also becomes important to model the mechanisms which she is using to acquire those concepts.

In summary we find that expert-based models of the pupil are inadequate. Modelling the learner in an open domain involves modelling a number of "concepts" which the learner has acquired and which cannot necessarily be built into the system in advance.

2.6-Nature of the interaction.

Most Tutoring systems operate a highly restricted form of interaction with the pupil. The restriction is not necessarily apparent since it is normally below the surface level. Systems such as SOPHIE appear to provide a reasonable natural language interface, but are actually very limited in the way in which they interact. The problem is that individual inputs from the pupil are treated separately - no attempt is made to integrate them into a conversation. It is as though everything before the current utterance is completely forgotten. This is clearly not the way in which a human teacher operates!

There are a few systems which attempt to maintain a structured interaction. A notable example is GUIDON. In this system there are problem-solving operators which derive a solution to the problem which teacher and pupil are discussing. The teacher then attempts to carry out an interaction which reflects the structure of the problem solution. SCHOLAR also produces a conversation structure. It uses importance-tags associated with the things it is able to talk about to select a set of goals and subgoals from which to choose the current topic of conversation. This work was based on an analysis of the behaviour of real teachers and, while it contains some ad-hoc sections, is actually capable of emulating the structure of some real teaching dialogues.

In general then, we wish a tutoring system to maintain a set of teaching goals which allow it to structure a discourse in order to make use of previous steps and to plan ahead. These goals do not form a uniform group, but may vary in nature.

Goals may be divided into those which remain static and those which are transient. A static goal is one which may never be completely satisfied. It is always present and influencing the interaction, although it may vary in importance. Each step in the interaction must be assessed in terms of the effect which it has had on each of these goals. Transient goals are ones which may be generated and, once satisfied, will cease to exist. It may be useful to introduce the idea of the scope of a goal, rather like the scope of a variable, which indicates the range of interaction for which a particular goal is valid.

There are several levels at which goals may exist. WEST, for example, identifies three levels: the first is basic mathematical skills, the second is the application of these skills in the WEST game environment, and the third consists of transferrable game-playing skills such as learning from your opponents moves. In general there seems to be a continuum of generality along which goals can be categorized.

The goal structure of a teacher corresponds to the set of goals which are currently active at a given time. There is clearly a link between this structure and the structure of the curriculum. They do not seem to be directly equivalent, but much of the goal structure may be derivable from the curriculum.

Another restriction which systems impose on the pupil is to permit only a very small set of responses at any given point in the interaction. Bobrow [Bobrow 1977] claims that most dialogue systems give a semblance of reasonable interaction by exercising close control over the dialogue. The opposite extreme to this would be a genuine mixed-initiative dialogue in which teacher and pupil may freely interrupt and change the topic of the dialogue.

Such interruption is easily handled in a reactive system which does not maintain a plan for the conversation, but systems which plan ahead must be capable of modifying their plans if an unexpected situation arises. We may describe the distance ahead which a system plans as a planning horizon for the interaction.

Any real interaction mechanism must make use of a model of the other participants in the dialogue. For a tutoring system this involves linking the pupil model to the interaction mechanisms. Such linking should affect both the content and the presentation style of the interaction.

In summary, a tutoring system should maintain a set of goals which corresponds to a plan for the interaction. These goals should be derived from the curriculum, the pupil model, and possibly some higher level sources. The goal structure should be flexible enough to permit modifications to be made if an unexpected dialogue situation arises.

2.7-Assessing the pupil.

Most Intelligent tutoring systems are based upon the sort of expert-modelling described above. This approach has a particular method of assessment within it. In essence the difference between the expert-model and the pupil-model is the current assessment of pupil progress. Unfortunately, rather than giving assessment the independant consideration which it deserves, most systems simply accept this approach as standard.

If we examine the literature of education, and work on LOGO, we find that assessment is a hotly debated issue. This problem should be reflected in tutoring system design. I propose that a major advantage of non-expert based modelling is that it forces the question of assessment to the forefront of design issues.

2.8-The problem of previous knowledge.

A pupil will not approach the system with an empty mind. There will be existing cognitive structures onto which the pupil will attempt to link the new material which she learns. The teacher should be capable of representing information about those preconceptions which the pupil is bringing to the domain since they may profoundly affect her knowledge of the domain, particularly at early stages. For instance, it is likely that it may be easier to link new items to existing knowledge than to other new items, so the pupil may represent the domain as a number of unconnected models which grow together over the course of time.

None of the tutoring systems which exist attempt to say anything about the problem of previous knowledge of the pupil. All assume that the student comes to them as a tabula rasa. BUGGY is a strange example of this because it assumes that no real world knowledge is brought to bear on the problem, but relies on the fact that certain subtraction bugs will have achieved stability due to earlier experience.

Summary of part2.

The nature of theories of education, and the need for a new theory to integrate computers with other educational media has been outlined. It has been suggested that a suitable starting point for this theory is the work of Jean-Jacques Rousseau. The educational methodology to which IMPART adheres has been outlined.

Brief sketches of some important Intelligent Tutoring Systems have been given and some general themes in tutoring system design have summarized. The importance of explicitly stating the educational philosophy associated with a particular system has been discussed. The issues of pupil modelling, choosing and representing a problem domain, and maintaining a structured interaction have been introduced.

Part 3 -

A tutoring system

for LISP.

Chapter 6.

Aims of the system.

1.-Design goals of the system.

In this section we will identify those aspects of tutoring system design which this project will pursue. These issues have been discussed earlier in the thesis; our purpose here is to provide a concise summary.

1.1-Overall goals.

A primary goal of the system is that it should embody a particular educational philosophy. We wish our pupils to learn to be self-suffcient in the application of a particular skill. We also wish to encourage the idiosyncratic exploration of our pupils, and will respect their freedom in sufficient degree to ensure that we do not impose constraints upon them without justification. The pupil should only respect the authority of our teacher in so far as the teacher earns it. Learning from experience and developing intrinsic motivation towards a subject will also be highly valued. Our focus will be upon the development of general skills rather than acquisition of specific knowledge, and our "curriculum" will provide a structure linking knowledge rather than a structure for the teaching interaction.

To explore this approach, our tutoring system will attempt to apply Guided Discovery Learning methods in an open domain. The system may be regarded as a "teacher" watching the interaction between the "pupil" and a "programming environment" while being prepared to intervene if it seems necessary.

The chosen domain is programming in LISP, which may be regarded as having a closed component (the syntax and semantics of the language), and an open component (the possible problem solutions). Because of limits on the time available for this work, the former part of the domain will be explored more completely, with some suggestions provided with regard to the latter part. The system should attempt to model and tutor three levels of knowledge: basic domain knowledge, rules for acquisition of that knowledge, and rules for the presentation of that knowledge.

To guide the learning process, the system will attempt to build upon the current state of the pupil's model of the domain. This involves taking a particular model of the learning process and applying it to observations which the pupil could make. This information will be used to guide a discourse and build a model of the current state of the pupil. Achieving this goal also involves attempting to represent those things which the pupil can directly perceive. Relating new information to this model precludes the arbitrary introduction of new topics by the teacher.

A consequence of attempting to build up from the current state of the pupil is that emphasis is placed on the techniques required to build problem solutions. It is not sufficient to simply present an algorithm to the pupil, there must be some jusitification of it in terms of things which the pupil already knows about.

The system does not possess a model of the "expert skills" of a programmer which it is trying to inculcate. The problems of an expert-based approach have been discussed above, and they are very apparent in the domain of programming where several researchers (e.g. [Schneiderman 1980]) have noted the vast discrepancies in the sort of skills possessed by individual "expert programmers".

No explicit representation of common programming errors will be incorporated into the system. Each problem which is discovered will be tackled using general rules and previous experience WITH THE CURRENT PUPIL. This also involves attempting to eliminate incorrect ideas before they have a chance to become stable in the pupil's mind. This results in a tutoring strategy which is an alternative to the expert/mal-rule based approach.

The interaction of pupil and teacher will have a structure which is derived from the current goals of the participants. The teacher will have a goal structure corresponding to a curriculum, which combines with information derived from the behaviour of the pupil to produce short and long-term order in the interaction. Tutoring rules which guide the form of interaction should be written in a domain independent manner.

It is important that the system should present arguments to the student which have been reasoned about at a semantic level. Deriving a problem solution by unguided search and presenting it without explanation will not aid the learner.

1.2-Simplifying assumptions.

Throughout the course of instruction the teacher should be at the side of the pupil to offer advice. If we may assume that this is the only aid which the student is receiving, then the user modelling may rely on the pupil knowing only those things which the teacher has taught. The model cannot be assumed to be perfect, but large rifts between the state of the pupil and the state of the model will not occur. This involves assuming an idealised world in which books and peer group discussions are avoided.

A related problem is that of real world knowledge. A pupil typically attempts to modify existing knowledge structures in order to model a new domain (the classic example being a model of a push-down stack based upon plate dispensers), rather than building something new. If a tutor has no knowledge outside the specialized domain then she cannot hope to model a pupil who is importing this type of knowledge, or to make use of the pre-existing knowledge structures of the pupil to enhance the educational interaction.

Because the computer tutor cannot represent all the real world knowledge, we assume that our student comes to us as a Tabula Rasa. This is a typical assumption of systems which attempt to model humans (e.g. [VanLehn 1981] , [Anderson 1982]), the normal argument being that if our domain is sufficiently abstract (as VanLehn claims subtraction is to high-school students) then so little world knowledge could be imported that the approximation is not unreasonable. Programming languages are probably sufficiently abstract to make this claim acceptable (though they could also be taught in a concrete way, using real world analogies).

A related argument is put forward by Papert in defence of LOGO [Papert 1980] . He claims that procedural knowledge is fundamental to all humans, and that things should be taught by fitting them to a procedural framework. Rather than trying to relate all our activities to this narrow base, I would propose that a valid use of computers is to lend concrete form to other sorts of abstract representation, thereby broadening the set of basic representations which the pupil may comfortably handle. When teaching Artificial Intelligence programming this means that a language should be taught as a new formalism, rather than by relating it to other things about which the pupil should know; we should attempt to instil a clear understanding of the language into our pupil WITHOUT recourse to analogies from other experience of the pupil.

The system is intended to be suitable for pupils learning Artificial Intelligence programming methods at postgraduate level. The students are expected to be novice computer users. The choice of audience permits the assumption that the students will be highly motivated to acquire LISP programming skills.

1.3-Architecture.

Because this system is intended to provide a framework for exploring the nature of interactive teaching, it is important to try to make each of the major constituents of the design into a separable unit in the implementation. Unpluggable units containing syntax, semantics and higher level constructs of a programming language should permit different languages to be taught. It is not (currently) possible to formalize the Educational Philosophy sufficiently to make it a separable part of the system, but making the teaching strategies unpluggable will go some way towards allowing an interchangable teaching style.

Chapter 7.

A toy example.

1.-A toy example.

This section outlines the teaching interaction which we are attempting to achieve. Its purpose is to give the reader an overall view of the problems facing the system, so not all details are explained. More complete descriptions of the design and the reasoning behind it are given in later sections. In order to make the role of each component clear a simplified world is used. The basic priciples to be demonstrated are the same ones that are applied in teaching a programming language, though not all features of the system can be shown in this world. In particular, all statements in this example are imperatives; there are no questions or conditional statements.

1.1-Choice of domain.

The toy world which has been selected is similar to that used by Richard Power in a program to generate conversation between two robots [Power 1979] . In this version, the pupil must learn to use a simple command language to achieve changes in the state of the world. The world consists of a room with a door which has a sliding bolt, and an "outside". There are two actors in the world, JOHN and MARY, each of whom may be either IN or OUT. To change from one state to another, an actor can MOVE, but nothing will happen unless the DOOR is OPEN. The DOOR may be OPEN or SHUT, and an ACTOR can move it from one state to the other by PUSHing it, although it will not change state unless the bolt is UP. The bolt can change state if an actor SLIDEs it, so long as the actor is

IN the room.

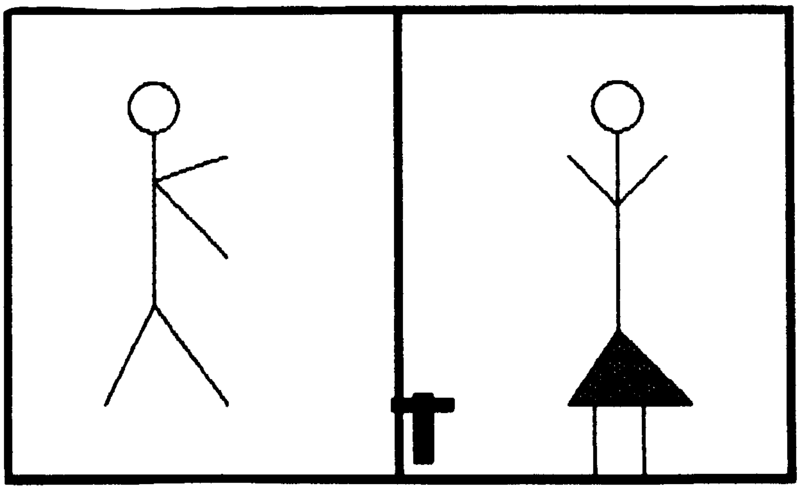


Figure 7 - A simple environment.

1.2-The command language.

In order to achieve something in this world, the pupil must issue statements in a simple command language. In this section we provide an informal description of the way in which knowledge about this language is represented.

1.2.1-Syntax.

The syntax of the language is simple since each possible action requires an actor and an object. The elements must be combined in the sequence

<ACTOR> <ACTION> <OBJECT>

e.g. MARY PUSH DOOR. Since the rectification of syntax errors is not the focus of interest of this tutoring system nothing more will be said about this aspect of the problem. It will be assumed that all input is syntactically correct.

1.2.2-Semantics.

The semantics of the language is described by producing a description of the effect of each action which is possible in the world. These descriptions consist of a list of tests which must be true before the action can be applied, and a list of modifications to make to the environment due to the execution of the action. These are called the preconditions, and the body, respectively. Formal descriptions of the semantics will be given later, but for the purposes of this example, the descriptions will be presented in English;

PUSH

Preconditions;

The first argument to push must be an ACTOR.

The second argument to push must be a member of the group of pushable objects.

Body;

If the BOLT is DOWN then do nothing.

If the object is OPEN, make a note that it is now SHUT.

If the object is SHUT, make a note that it is now OPEN.

SLIDE

Preconditions;

The first argument to slide is an ACTOR who is IN.

The second argument to slide is a member of the group of slidable objects.

Body;

If the object is UP, make a note that it is now DOWN.

If the object is DOWN, make a note that it is now UP.

MOVE

Preconditions;

The first argument to move is an ACTOR.

The second argument to move is the same as the first.

Body;

If the DOOR is SHUT then do nothing.

If the ACTOR is in, make a note that it is now OUT.

If the ACTOR is out, make a note that it is now in.

Figure 8 - Toy language semantics.

1.3-Model of the pupil.

As was mentioned earlier, we wish to model the pupil in terms of a learning theory applied to her perception of the environment. In this case our system must note which language statements the pupil has used, and must attempt to derive the set of hypotheses which the student could hold about their behaviour. The system should also monitor the meta-level of the pupil's ability to respond to error messages.

1.4-Goals of the teacher.

The major goals of the teacher are to get the pupil familiar with the command language, and with the programming environment. Within this, the system seeks to extend the pupils knowledge of the language statements, of methods of combining them, and of problem-solving strategies.

1.5-A sample interaction.

The following interaction is intended to give some idea of the role which the computer would fulfill in a teaching situation. Actual steps in the interaction appear in numbered boxes, with the originator indicated as TEACHER or PUPIL. These are interspersed with descriptions of the processes which are occuring within the teacher.

The "natural language" flavour of the interaction is deceptive. Output is "canned" or generated using unprincipled tricks, and input is in a very restricted subset of English, or may be produced by non-linguistic devices such as pointing to items on the screen.

==============================

```
*********************************************************************
*  0:  The pupil sits down at the machine, types her name, and is     *
* presented with a screen which shows three menus, a view of the      *
* world, and a region for talking to the teacher.  One menu contains  *
* the editing functions for generating commands, a second contains    *
* the list of possible actions in the world, and the third contains   *
* the list of objects which exist in the world.  There is also a      *
* message inviting new users to type HELP.}                           *
*********************************************************************
```

The problem of designing the user interface is a complex and interesting one, but is beyond the scope of this program. All the display environments and "tools" for interacting with the world are generated in advance. It is also necessary to provide canned explanation for those aspects which are beyond the knowledge of the system.

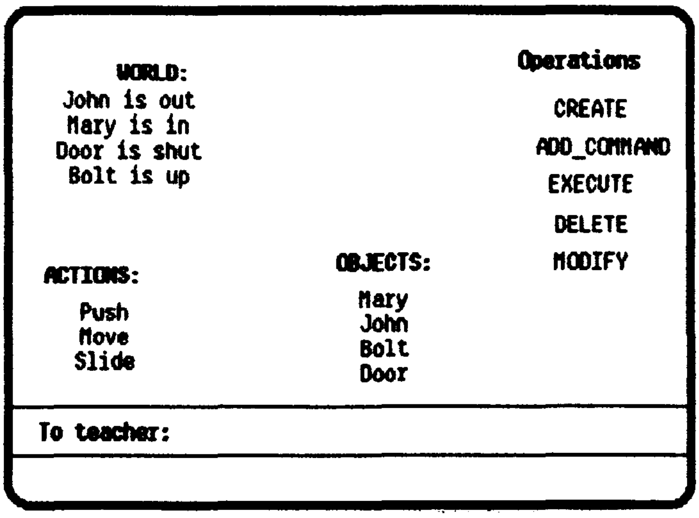


Figure 9 - Toy screen layout.

****

* 1: PUPIL: Help. *

****

The teacher initialises a model of a new pupil who knows nothing about LISP. In response to the request for help, a canned speech is produced by the "introduction" topic controller.

****

* 2: TEACHER: On the screen there are three menus - editing functions, *
* actions and objects. Your goal is to make things happen in the world *
* by creating commands using actions and objects. To do this you use   *
* editing functions. Point to CREATE to generate a command, then try   *
* it out using EXECUTE.   *

****

****

* 3: PUPIL: Points to CREATE, then MOVE, resulting in the template   *
* ~ACTOR~ MOVE ~OBJECT~. Pupil asks teacher "What does ~actor~ mean?" *

****

The question is about one particular slot in one particular template. Its effect is to activate topic controllers to do with actions, arguments and datatypes which talk at a general level using specific Illustrations from the current expression. Each of these contollers contributes to the following speech.

```
*****************************************************************************
*  4: TEACHER:  MOVE is an action.  It has two ARGUMENTS.  ARGUMENT 1 is *
* is an ~ACTOR~.  This can be JOHN or MARY.  ARGUMENT2 is an ~OBJECT~.   *
* This can be anything on the object menu.                                *
*****************************************************************************
```

```
*************************************************************************
*  5: PUPIL:  Fills in template to produce JOHN MOVE MARY.  She has  *
* forgotten about executing it so asks "What next?".                  *
*************************************************************************
```

The teacher has already evaluated this expression and identified the errors which will result from executing it. It could tutor the errors now, but a tutoring rule makes it refrain until the expression has been evaluated by the pupil. It recalls the intention of generating and executing an expression, so a topic controller reminds the pupil about the EXECUTE operation.

```
****************************************************************************
* 6: TEACHER: If you use the EXECUTE operation, you can see what effect *
* your command has on the world.                                         *
****************************************************************************
```

```
****************************************************************************
*  7: WORLD: Pupil points to EXECUTE, and the command JOHN MOVE MARY is *
* carried out.  The message "JOHN cannot MOVE MARY" results.             *
****************************************************************************
```

Referring back to the description of MOVE given in figure 8, we see that the semantic requirements have been violated since MOVE must have the same object as both arguments. Now that the execution has been performed the datatype topic controller can be applied to the description of MOVE, producing a speech which clarifies the error. The problem-solver is used to suggest a change to the expression, and this invokes the MODIFY topic controller to introduce a new editing operation.

```
************************************************************************
*  8: TEACHER: This statement did not succeed, since the first and     *
* second argument to MOVE must be the same, and must be an ACTOR. You  *
* can change your command by using MODIFY or you can CREATE a new one. *
************************************************************************
```

```
*************************************************************************
*  9: WORLD: Pupil modifies command to JOHN PUSH JOHN, and executes it. *
* World reports "nothing has changed".                                  *
*************************************************************************
```

The teacher notes that execute has been used unprompted. This message was expected by the teacher, and a speech is generated to amplify the reasons for this behaviour. In effect the speech is simply a translation into english of the path which was followed through the semantic description. The tutor uses the problem-solver to find a situation in which an alternative course of action would be followed, and tells the pupil about it.

```
*************************************************************************
*  10: TEACHER: The reason that nothing has changed in the world is that*
* the MOVE command does nothing if the door is shut.  You can open the  *
* door by PUSHing it.                                                   *
*************************************************************************
```

Chapter 8.

A syntax-directed

programming environment.

1.-The programming environment - MATILDA.

1.1-INTRODUCTION

The initial problem was to design a programming environment which would facilitate the learning of the programming language LISP. The Users were expected to be novices with no previous computing experience. The environment was not intended for autonomous use, but to be part of a course including individual tuition, a reference text [Winston 1981] , and group discussions about the language. No lectures were given, and the text was not introduced until several weeks into the course owing to differences in dialect and method of approach.

The intention of the programming is to provide people with an understanding of some fundamental concepts about programming and computers, without necessarily producing expert programmers. One consequence of this is that knowledge of the hardware of the machines and of operational details (such as disk file handling) is not expected of the students, and may in fact be a distraction from the main purpose of the course. This is manifested as a constraint upon the interface to protect the user from this side of the system. Interest is focussed upon the virtual machine represented by a LISP interpreter.

In order to preserve the consistency of the language which is being learnt (and hopefully make it easier to learn), a fairly pure version of LISP is used which does not involve global variables and retains as few side effects as is practicable. These features are introduced later in the course, when this teaching environment is no longer in use, for people who wish to write large programs.

MATILDA was expected to provide a tool which made it reasonable for students to learn about LISP by experimenting with expressions in the language. To this end, everything that the User does to the system should result in some sort of constructive response.

1.2-THE MAIN FEATURES OF THE ENVIRONMENT.

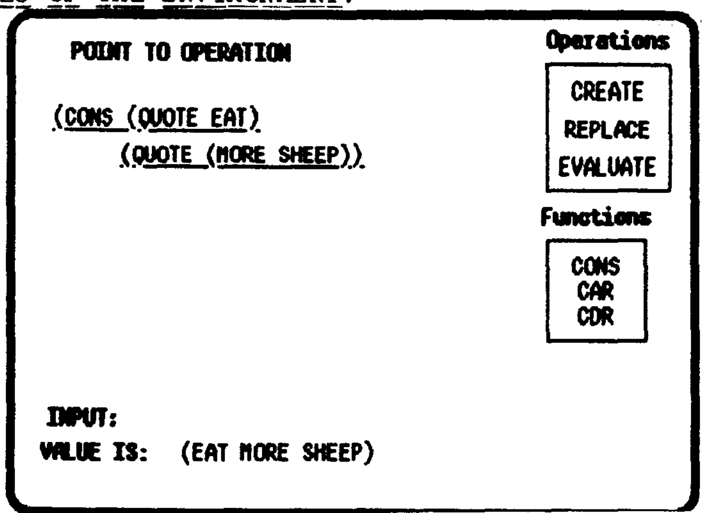


Figure 10 - MATILDA - screen layout.

As seen by the User, MATILDA consists of three menus from which items can be selected by pointing and a region of screen in which a LISP expression can be built. The menus contain editing commands (menu 1), functions available in LISP (menu 2), and functions defined by the User (menu 3). There are also three smaller regions in which information appears. One region is for users to type input, one is for values returned from LISP, and one is for messages from the system. Use of the system involves pointing to Editing Operations, and then specifying arguments for those operations by pointing to function names and to Locations in the expression on which the User is working. When the expression is completed to the user's satisfaction she may transmit it to the LISP interpreter and receive information about its behaviour. This explicit division between the interface and the language was intentional, its purpose being to ease the process of transfer to a normal interpreter at a later date.

1.2.1-CURRENT EXPRESSION

The expression on which the user is working is maintained on the screen until it is explicitly dismissed by the User. This expression provides an external representation of the user's focus of attention. The expression can be modified incrementally and evaluated repeatedly in order to examine the relevance of particular parts to the overall effect. This makes it easier for users to test and correct their hypotheses and examine conditions under which they break down.

Since an expression typed to a normal interpreter is lost as soon as it has been evaluated, students tend to avoid writing long expressions or making incremental modifications in order to reduce typing. This situation can be improved by making all expressions into function definitions, but this involves introducing the concepts embodied in defining functions before the basics of the language can be thoroughly understood. The Current Expression in MATILDA overcomes this difficulty and means that function definitions need not be introduced until much later than is customary.

1.2.2-MENUS

The menus are regions of the screen which display inputs that the user can make (such as LISP function names). Any item on a menu can be indicated to be the next input by pointing to it. This serves the purpose of freeing users who cannot type from learning this skill concurrently with learning the language. By explicitly displaying the operations and functions available to the User it is possible to ensure that she is aware of all the options open to her. The inputs available to the user are divided into two categories, which are represented by two separate menus.

The first menu (menu 1) contains editing operations which are recognised by the interface. As soon as the user points to one of these it is executed, prompting the user for any arguments which are necessary. This separation of the interface operations from Lisp functions is intended to encourage the user to regard Lisp as a separate entity from the interface for reasons mentioned earlier. Some of these operations will be briefly described.

CREATE clears whatever Current Expression previously existed, and starts a new one. It prompts the user to point to a function name, and this becomes the new Current Expression.

REPLACE is used to fill in a slot in a Current Expression. The User is asked to indicate first the slot to be filled, then the function with which to fill it.

EVALUATE passes the Current Expression to the Lisp interpreter for evaluation, and displays the resulting value or error message.

Menu 2 contains the names of the functions available in LISP. Pointing to an item on this menu results in that function appearing on the screen in a syntactically correct format with "slots" marking the parts which must be filled in to form a valid expression.

The final menu (menu 3) contains the names of user-defined functions. These appear as soon as the function name and number of arguments has been specified, since this is sufficient information to allow the system to assign a default template for it. It is necessary for these names to become available before the function is fully defined in order to facilitate recursive function definitions.

1.2.3-TEMPLATES

When a function call is requested as part of a Current Expression, a syntactically correct calling format for that function appears on the screen. This includes the appropriate parentheses, the correct number of unoccupied slots for the arguments, and an indicator of the type of expression required for each slot. This eliminates trivial syntactic problems such as miscounting of brackets, which do not throw any light on the level of understanding of the individual and can be very frustrating. The fact that a complete function call is the smallest element appearing on the screen is intended to encourage the User to regard this as the basic entity of LISP.

1.2.4-MICROWORLD SEQUENCE

Rather than making all possible LISP functions accessible to the User on the first occasion that she encounters the interface, the language is introduced through a series of subsets of functions called Microworlds (in analogy with LOGO [Papert 1978]). The functions in these microworlds are intended to be suitable for examination by the User (i.e. she is thought to have satisfied all the prerequisites for understanding them), and are chosen to guide her on a reasonable course to learning the language. In this case the sequence started with list building functions, then those for dissecting lists, then predicates and conditionals and finally function definitions. It was not intended that this sequence be rigorously adhered to, and microworlds have been generated to reflect individual interests.

1.2.5-STRUCTURAL CURSOR CONTROLS

When the student must indicate a particular location within the Current Expression she may use the cursor keys to manipulate an extended Cursor on the expression. This Cursor always highlights a complete subexpression, and the keys affect it's position in a manner which reflects the structure of the expression: Left and Right arrows move along a list, Down descends into a list, and Up moves to the list containing the Current Location. It is hoped that this practical outlet for knowledge about expression structure will encourage students to regard a LISP Expression as a structured object rather than a piece of homogeneous text.

1.2.6-FEEDBACK

Enabling people to understand what is occuring when an expression is passed to the LISP interpreter requires that they be presented with information that helps them to build a model of the actions taking place. Unfortunately, constraints of time meant that most of the feedback information planned for MATILDA has not been implemented. The only information is the Evaluation results and Error Messages from the interpreter itself.

1.3-OBSERVED USE OF THE SYSTEM.

Students used the system for between 5 weeks and 3 months before moving to use of an unadorned interpreter. This section is based upon general observations and comments made by the students during this time, together with the results of a Questionnaire produced by Roy Pinder at Warwick University. Some global observations are followed by specific comments about each of the features described above. The only source of information for comparison is experience teaching students from the previous year, who used a normal interpreter.

1.3.1-DIFFICULTIES DUE TO IMPLEMENTATION

A Universal criticism of the system was the low speed with which menus could be scrolled to select items. This was an implementation problem due to the fact that special screen handling extensions produced for the LISP interpreter operated via CP/M in order to preserve compatability across most Z80 based systems. Subsequent versions of MATILDA have better display handling, but these have not yet been made available to students. This constraint was a source of frustration which resulted in many users moving onto an unadorned interpreter earlier than they would otherwise have done. There were also problems with the reliability of the hardware used for teaching which detracted from the intended simplicity of the interaction. These have now been rectified.

1.3.2-FREE LEARNING

The intent of providing a safe environment in which people could discover LISP by experiment was not fully realised. A major part of the problem was that many people felt unhappy about just trying things in order to discover how they behave. The question "What shall I do now?" was frequent in early use of the system. The introduction of discussion sessions to share knowledge about the language did not noticably improve the level of this non-directed exploration.

In part, this can be attributed to the fact that goals in LISP are very different from those in domains of which the students had previous experience. Producing an expression which returns NIL is not inherently exciting. One method for overcoming this discrepancy is to start from a subset of LISP which has goals about which the student has some preconceptions, such as numeric functions. This approach is used in Winston's book [Winston 1981] , and was tried with one student who was particularly uncomfortable with the other microworlds. An alternative is to provide a worksheet of problems which specify lisp-like goals and require students to find expressions to return the appropriate values. These can become progressively less restrictive until the students generate their own subgoals. A further technique would be to provide a graphical interpretation of the action of functions in transforming arguments into values. This would help increase the importance which the user assigns to the value.

A further origin of this difficulty is that since free learning is not encouraged by the educational system in this country it has to be relearnt by students at this level. The people who achieved most with MATILDA equated it with a Video Game. A more difficult problem to deal with was that some people carried on interactions on a basis akin to random keypresses, and kept this up for a long time without actually learning anything. It seems that this must be handled by direct teaching of the learning skills.

1.3.3-CONCEPTUAL UNDERSTANDING

The level of understanding of fundamental concepts was gauged from the group discussion sessions. A high level of understanding was displayed on all topics discussed. Many concepts discovered at an early date with this system (such as the question of evaluation order) were never discovered by the previous group of students. This was reflected in the fact that all students could make sensible suggestions about how to tackle new problems.

1.3.4-CURRENT EXPRESSION

No comments upon the idea of the Current Expression were made while people were working with Matilda, but the lack of such an expression was one of the first difficulties reported when students moved to the interpreter. Structure editors or external screen editors came into use to fill this role.

1.3.5-MENUS

Apart from the problem with the speed of operation, the menus were very popular - both as a way of seeing the options which were available and as a means of reducing typing. When the option of typing function names rather than pointing was introduced it was only used for functions which were widely separated on the menu.

1.3.6-TEMPLATES

Most students commented on the usefulness of the syntactic information provided by the templates. On transferring to a simple interpreter, the number of syntactic errors which were observed was considerably lower than the error rate of students who had never used MATILDA. It is proposed that this is due to the early development of a method of analysing expressions which makes use of the underlying semantic information.

1.3.7-MICROWORLD SEQUENCE

The sequence of microworlds was found to be generally sucessful. The early worlds were perhaps too small, since they were only incremented by one or two functions at a time. Each of these microworlds was used for a period of about two hours. The very first environment was not a good idea in practice. It consisted of EVAL and QUOTE, but since no side-effects were admitted, no variables could have values, so it was difficult to produce an expression which evaluated to anything other than NIL. This lack of side effects caused other problems which will be discussed below.

1.3.8-CURSOR CONTROLS

The structural Cursor controls achieved their purpose in a very convincing manner. The dependence of some concrete goal such as moving round an expression upon modelling the somewhat abstract structure of the expressions quickly led to a complete understanding of this means of regarding the language. All the students are able to focus attention on subexpressions, and none had difficulty in learning to use an INTERLISP type structure editor. This is in direct contrast to the problems experienced by the previous year's students in this respect.

1.3.9-FEEDBACK

As was expected, the limited feedback provided by the system at the time that it was introduced for use resulted in problems for the learners. The individual tutorials allowed this to be dealt with by explanations of the information that the system would have returned, but this is far from satisfactory. Most major of the problems was the cryptic nature of the error messages which were returned direct from the interpreter and had a very low information content; messages were often completely inappropriate for the error which caused them. Another source of feedback which would be very useful is a package to give information about intermediate stages in evaluation (i.e. a trace package).

1.4-SUMMARY.

Experience gained with MATILDA as a stand-alone system suggests that

Discovery Learning is an excellent way to attain understanding of a subject area provided that it is used in conjunction with some guidance which attempts to overcome the motivational difficulties of the user and maintain a reassuring interaction. Powerful feedback to aid the detection and correction of errors (both within the program and within the thought processes which led to the program) is also necessary. With MATILDA this was provided by a teacher. IMPART is an attempt to provide this aid within the machine which is in one-to-one contact with the student.

2.-Representing syntax in IMPART.

The assumption made by this system is that a syntactically incorrect program is not a program at all, and as such is beyond the scope of the tutor. From experience with MATILDA it seems that students who never see syntactically incorrect expressions learn the structure of correct expressions and generate few errorful expressions when free to do so. It is suggested that this is because they learn to associate a particular syntactic structure with an underlying semantic form, and hence find syntax errors very obvious. This view is supported by empirical work carried out by Weiser [Weiser 1982] and Shneiderman [Shneiderman 1980] .

In order to ensure that the tutor and pupil see only syntactically correct programs, the pupil interacts with the programming language via a syntax directed editor. The editor is based upon MATILDA, a Menu and Template based editor for LISP which has been discussed above.

All language statements and editor operations are made visible on menus so that the options available to the student at any time are explicit. When a function is selected, a syntactically correct template for that function is displayed. The pupil may now fill in the arguments (unfilled slots) of the template. Moving around the expression is achieved by issuing commands which reflect the underlying structure of the expression (for example, UP would move the cursor to the parent node of the expression it is currently on) rather than the surface appearance of the text.

2.1-Syntax editor.

The original version of MATILDA was specific to LISP. In order to try and achieve domain independence in the current system a new screen-based structure editor was devised which has a plug-in definition of the syntax of the language. It was originally hoped to derive the editor behaviour from a Backus-Naur form description, but in practice a more specific description has been used. Backus-Naur based editors such as EMILY [Hansen 1971] and other forms of language independent editor (see e.g. [Medina-Mora 1982]) are now well understood. It seems likely that the translation from BNF to the representation used in this editor will be an easily mechanizable task.

The editor must display expressions for the pupil in terms of the agreed syntax of the language. The other components of IMPART operate upon a parse tree of the expression. The editor must be capable of transforming expressions to parse-trees and vice versa if it is to provide a useful interface between the pupil and the rest of the system. In fact, since this is the only point in the system which embodies syntactic knowledge, all expressions in the target language (including, for example, illustrations used by the teacher during conversation) must pass through the editor.

As an example, suppose the pupil wishes to create the expression

(CONS (QUOTE A) (QUOTE (B C))

The instruction sequence CREATE CONS produces the template

(CONS ~arg1~ ~arg2~)

The pupil may REPLACE each argument with QUOTE, and REPLACE the arguments to QUOTE with the appropriate text. The completed expression will be passed to the rest of the system in the parsed form;

```
cons(quote(a),quote(**list(b,c)))
```

and the result passed to the editor will be

```
**list(a,b,c)
```

which will be displayed as

```
(A B C)
```

The editor is driven by pointing to items on menus - this applies to editor operations and program statements. The screen layout used by this tutoring system is shown in figure 11.

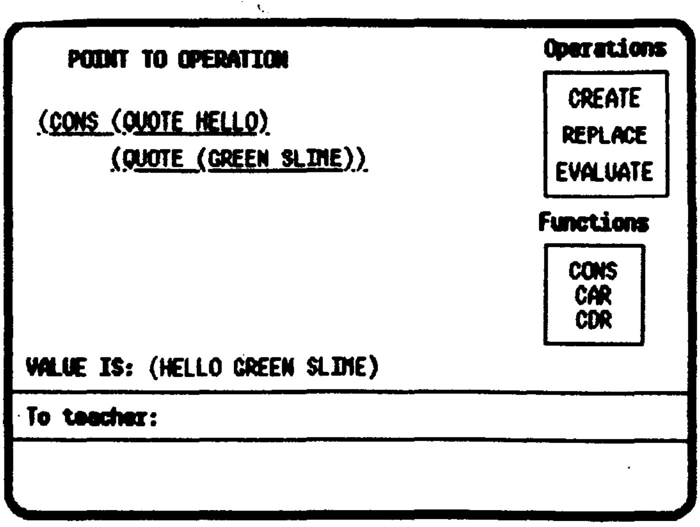


Figure 11 - Overall screen layout.

Figure 12 illustrates the relationship of the programming environment to the overall structure of IMPART. MATILDA may be regarded as a non-intelligent component which is part of the environment rather than the teacher.

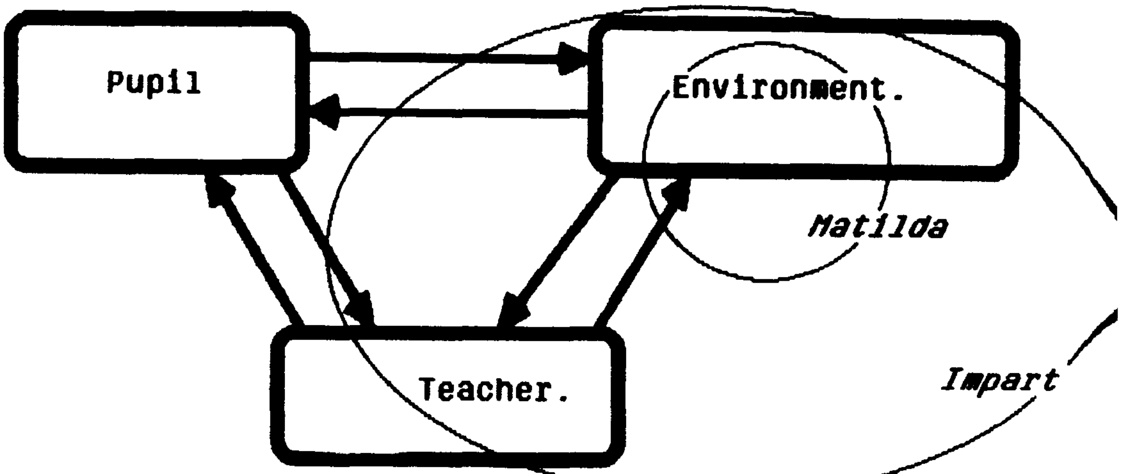


Figure 12 - The relationship of MATILDA to IMPART.

3.-Summary of chapter 8.

The design of a syntax-based programming environment called MATILDA has been discussed, and experience with the use of this environment has been reported. A more general syntax-directed editor used in IMPART has been briefly outlined. The role of this component in relation to the rest of the system has been discussed.

Chapter 9.

Choosing a representation

for

program semantics.

In order to teach a programming language, the system must have knowledge about the "meaning" of statements in the language. Since there exist well developed theories of the semantics of formal languages, it was decided to assess these as possible forms of representation. This section outlines the problems to which the semantic representation must be applied and briefly summarises the difficulties associated with three standard approaches to computer language semantics. Following this, the reasons for choosing the formalism which was finally adopted are given.

1.-Use to which semantic representation will be put.

In deciding upon a way to represent the programming language which is to be taught, it is important to consider the way in which the representation will be used. This section indicates those tasks in the system which make use of the semantic description and attempts to point out the requirements which each task places on the representation.

It is important to consider whether the activities which the system must support can be dealt with effectively by a single form of the semantics, or whether multiple representations should be maintained. Separate semantic descriptions have the advantage that their form can be adapted to maximise their efficiency for a particular application, but it is necessary to carry out more maintainance if a modification is made. There is also a problem in demonstrating the consistency of different representations. The possibilities of multiple formal semantic systems being combined to produce an internally consistent group of language definitions has been investigated in the past [Hoare 1974] . In the present system a decision was made to carry the use of a single representation as far as possible, being prepared to change to multiple descriptions if insuperable problems were met.

To distinguish the language being taught from the implementation languages, it will subsequently be described as the target language of the system. This also serves as a reminder that the system should attempt to tutor several different languages.

1.1-Executing a program.

The first requirement of the system is that it should be capable of deriving the correct behaviour for any expression in the target language. The initial intention was to include a normal language interpreter in the system, and have an independent semantic representation which corresponded with it. In fact, following the above discussion of duplicated information in the representations it was decided to use a single semantic representation as the ONLY means of executing expressions in the system. This representation must be usable to produce input/output behaviour like that of a real interpreter, and must respond to violations of the semantics with "error messages". These violations are only one form of error recognised by the teacher. Other sorts of error will be discussed later.

The teacher must be aware of errors generated by the pupil as soon as they occur if it is to decide how to tutor each problem in the most appropriate way. For this reason it is important that it should be possible to execute INCOMPLETE expressions in such a way as to identify errors in completed sections of the expression, while ignoring errors which are caused by incompleteness. For example, given the program segment

```
vars xy:integer;

 while x<y do
```

it should be immediately apparent to the system that some undeclared variables have been used and that the underlying intention was probably to declare them in the earlier statement. Both this and the lack of a second argument to the "while do" statement cause violations of the semantic description of the statement, but the latter problem should not be tutored since it is due to incompleteness. A special case of this is permitting the execution of subexpressions of a complete expression, by generating an appropriate context in which they can operate. This is valuable in allowing pupil or teacher to reduce a problem to its essential features. As an example, it would be useful to demonstrate the action of Pascal loop statements by executing the expression

```
for i:=1 to 5 do writeln(i)
```

whereas in actual fact, the code necessary to execute this is

```
program fred(input,output);

    var i:integer;

    begin
       for i:=1 to 5 do writeln(i)
    end.
```

A necessary extension of the error detecting facilities would be to provide a system which finds all the manifestations of a particular error, and similarly identifies all other errors and their manifestations. If all this information regarding potential errors is available to the teacher, she can take decisions about which error to deal with first, and which manifestation of it to concentrate on. This decision can be taken with regard to the state of the student and the teaching techniques available. For instance, consider the expression;

```
(APPEND (PLUS 3 X)
        (CAR Z))
```

where X is 'A and Z is '(B C D),

The first error detected during normal evaluation would be that A is an inappropriate argument to PLUS, so the whole expression would result in an error message. If this error is corrected, we should then find that whatever the result of PLUS, it is an inappropriate first argument to APPEND. Correcting this would show that B (the result of (CAR Z)) is also an inappropriate argument to APPEND. This information is necessary as a basis for deciding what to tell the pupil. It is worth noting that the inappropriateness of these arguments to APPEND is for different reasons. PLUS is always inappropriate, but CAR is correct for some values of Z.

The language interpreter (which in this case is a role filled by the semantic description), should behave in a way which will enable pupils to progress to a normal programming environment when they leave the system. For this reason, it is necessary to generate error messages to be given to the user. In general, these will be supported by further explanation from the teacher. The semantics should provide the basis for a system of simple error messages which are not misleading, though they may be less informative than is optimal in a teaching environment. These simple messages should be produced in parallel with more detailed error specifications which are passed to the teacher.

Part of the difficulty experienced in learning programming languages is to do with perceiving what happens during execution of an expression with sufficient clarity to model it. This precedes the problem of understanding what has happened. To provide this facility, it is important to be able to see the way in which the environment is changed by individual expressions and small groups of expressions. For this reason, there is a requirement that the semantic representation can be accessed in a way that enables effects to be made "visible" as they happen. This is a more general form of the sort of task which is traditionally filled by tools such as a trace package.

1.2-Describing statements and programs.

At various points during the interaction, the teacher will need to describe all or part of the action of a statement to the pupil. The depth and form of the explanation will vary depending on the knowledge of the user. For this reason, a semantic representation which can be used to generate psychologically reasonable descriptions of varying levels of complexity is very important.

These descriptions will be used in two ways. The first is in generating complete descriptions of the behaviour of a single statement. This fulfills a role rather like that of a help package, although it must be "intelligent" in the sense of adapting the form of explanation to the level of knowledge of the pupil.

The second, and more complex, form of description which is necessary is the partial description of the action of programs. This involves the same techniques as the first usage, but also requires the presence of some knowledge to enable the system to select only those parts of the semantics which are relevant to the current discussion, and blend them into a cohesive whole. In particular, this will require being able to abstract common parts from several different statements within a complete expression. Many different problems arise with this application, as the following example illustrates;

```
(DE LENGTH (L)
   (COND ((NULL L) 0)
         (T (ADD1 (LENGTH (CDR L))))))
```

If we wish to discuss the way in which the recursion progresses, it is necessary to monitor the values of L and the values returned by the call to LENGTH. If we wished to discuss the way in which the problem is decomposed, we would also introduce the values returned by CDR. A discussion of the way in which results are built involves the value of ADD1. The semantics should provide a mechanism which supports these forms of "focussed analysis" of an expression.

1.3-Relation to general issues.

A teacher does not always talk about specific instances of a problem. A goal of teaching must be to provide the pupil with the most general lesson which can be learnt from a particular experience. For example, if the student evaluates;

```
(CAR X)
```

where X is unbound,

then a direct translation from the semantic error to English would be something like "CAR attempts to find the value of X and take the first element of that value. X does not have a value". A teacher would probably say something like "Most lisp functions evaluate their arguments before doing anything to them. Before CAR takes the first element of X it tries to evaluate it, but fails because X does not have a value". The latter explanation offers more information to the pupil by relating the case to other instances.

Achieving this sort of generalization requires knowledge about reasonable forms of comparison to draw to the attention of the user. There should be "rules of extrapolation" which can decide what level of information to give to the pupil. This decision should be taken by examining the entire language description and comparing it to the particular case being studied. For example, drawing an analogy with SETQ would be wholly inappropriate in the case shown above. The semantic description should facilitate such extrapolation from particular instances.

1.4-Evolution of larger programming primitives.

It has been pointed out that experts in a particular domain often perceive that domain differently from novices. This has been observed to hold for computer programming [Schneiderman 1979] and summarized as follows;

> " An expert computer programmer encodes and processes information semantically, ignoring programming language syntactic details." [Weiser 1982]

Weiser introduces the concept of a "slice" to describe a programmers perception of a program. This is a non-contigous section of code within a program containing all the expressions which could affect a particular variable at a particular statement. Weiser claims that the process of debugging a program involves a programmer in making a mental "slice" from the point at which the error occured in order to restrict the possible locations of a cause of error. The validity of this view has been demonstrated empirically [Weiser 1982] , and an automated slicing algorithm has been developed. A claim is made that slicing techniques should be taught as part of programming;

> " Slicing is now reinvented by every programmer who uses it. Beginning programmers taught the concept of slicing could avoid this reinvention and could more rapidly improve their debugging skills."

In this and other work on program decomposition [Zislis 1975] the emphasis is upon dividing a program into sections based upon dataflow with the goal of aiding debugging. I would propose that the technique of slicing is more generally applicable than Weiser claims. The mechanisms for achieving semantic decomposition of a program are a fundamental part of establishing a relationship between high level algorithms and the semantics of individual statements: There exist inverses of slicing methods which guide the process of problem solution.

Since a goal of the system design is to build upon the perceptions of the pupil it seems important to provide a framework which is psychologically meaningful for building programs out of primitives. Slicing provides a basis for that framework. To this end we should attempt to choose a semantic formalism which will support the use of slicing techniques to aid debugging, and the inverses of these techniques to aid program design.

1.5-Problem solving and generation.

When the pupil is attempting to solve a problem, the teacher must be aware of at least one correct solution to the problem in order to make judgements about the amount of progress which the learner is making. This involves knowing what the pupil is trying to achieve, and deriving a solution path from this information together with knowledge of the language. In general, the problem solution should be derived in relation to the user model so that it will be similar in form to the pupils own solution. Since this system is intended to support mixed-initiative interaction, problems may be generated by the machine or the student. How these problems are specified may vary.

One method of specification is to describe a desired final state, and give an initial state from which the final state must be achieved. This is the approach used in the TOY EXAMPLE (chapter 7), and results in descriptions such as " Try to get John inside" (final state). The initial state is explicitly given on the screen in this case. A possible solution generated by the program would be;

```
MARY SLIDE BOLT
JOHN PUSH DOOR
```

JOHN MOVE JOHN

The toy version of the system generates this solution by unguided depth first search. Such a technique would quickly run into problems of combinatorial explosion if it was applied to any programming language of reasonable size, so some form of constraint upon the search is necessary. There is a large literature on program proving and some techniques may be borrowed from this area, although the restricted nature of the problems tackled by novices are amenable to less powerful techniques.

In the course of tutoring, the teacher needs to supply the pupil with example problems relevant to the current work, and exercises which the pupil may try in order to practice new skills. For this reason, it is necessary to provide some means of combining the semantic descriptions with a "specification of difficulty" in order to generate such problems. The specification of difficulty will presumably be derived from the user model.

The overall aim is to produce a program which takes a problem specification as input and generates a program which will solve the problem in the original specification. This is the basic goal of automatic programming systems.

To deal with the generation task, we wish to be able to supply the program with a partial specification of a problem (i.e. the set of things which we wish the pupil to use). In this case our program should produce a problem solution, and a more complete instantiation of the problem specification. For example, we wish to be able to specify that we require a problem to test the pupil's use of list-accessing functions. In return we wish our problem generator/solver to produce the problem; "Write an expression to extract the third element of the list (MANDY EATS BURGERS FREQUENTLY)" and the solution; (CAR(CDR(CDR(QUOTE (MANDY EATS BURGERS FREQUENTLY)))))

In the context of a tutoring system, there are various constraints at work. One of these is that the solution should be understandable to the pupil. This involves ensuring that no techniques are used which are beyond the level of programming ability which the student has achieved.

The methods which the problem-solver used to generate the solution must be documented alongside the solution so that they can be used as the basis of instructional dialogue by the teacher if that proves necessary.

An important issue here is the glass-box or black-box nature of the problem-solver. It is implicit in the above description that some of the workings of the problem-solver should be made visible to the pupil. It is not the case, however, that we wish to regard the problem-solver as our programming expert and attempt to make our pupil emulate this problem-solving style. Instead, we wish the problem-solver to adapt to the pupil. The fundamental components of the problem-solving strategies should be derived from the model of the pupil. The only built-in components will be of such a general nature as to be unquestionably necessary for any problem solution.

2.-Choice of formalism.

Since the 1950's there have been attempts to describe computer language behaviour in terms of a formal representation. This task has been attempted for a variety of reasons, such as specifying language behaviour across installations, proving facts about individual programs (such as whether they will terminate) and examining theoretical problems in computer science, such as computability. This was summarized by Hoare as follows;

> " The objective of a formal description of a programming language is to give a clear and unambiguous definition of the interface between the designers, users and implementors of the language, which they can all understand and refer to." [Hoare 1974]

Backus-Naur form has become a universally accepted way to describe programming language syntax. There has not been an approach to semantics which has had such success. The three most influential attempts have been Operational, Axiomatic and Denotational semantics.

In this section, a brief summary will be given of the limits of each of the existing formalisms. This will be followed by an overview of the semantic representation which has actually been adopted.

2.1-Applicability of formal language semantics.

2.1.1-Operational semantics.

2.1.1.1-General difficulties.

There are several limitations of this form of semantic representation.

An important theoretical point is that this representation does not resolve the question of the "meaning" of an abstract expression. Complex expressions are reduced to a large sequence of simpler expressions, but ultimately we reach the level of the basic operations, and we must rely on our own understanding to endow these with meaning. Operational semantics reduces the behaviour of a complex machine to the behaviour of a machine so simple that it could not possibly be misunderstood.

A program is basically described by mimicing what happens during its execution. For this reason the semantics are not useful for making decisions about whether or not a program will terminate. Any program which locks itself into an infinite loop will have a corresponding operational semantic description which locks itself into an infinite loop. This was not really a problem in the original design of these semantics, since they were intended as a guide to the necessary behaviour of an implementation of a language rather than a system for proving facts about various properties of specific programs.

A further consequence of this method of modelling programming semantics is that the operational approach does not allow statements to be made about expected outcomes or relationships between states. This makes the representation unsuitable for such tasks as automatic programming or program specification.

The control flow between statements is a feature of the operational semantic machine. It is effectively a direct semantics, unable to deal with jumps and labels.

2.1.1.2-Application to tutoring.

Any form of operational semantics relies heavily on the architecture of its own abstract machine in reducing the "meaning" of an expression to a problem with simpler units. It has been noted that the teacher must be prepared to make drastic variations in the way she talks about a problem domain in order to make those descriptions more easily assimilable by the pupil. This flexibility is NOT easily achievable using Operational semantics. The systems which communicate with the pupil would need to be capable of drastic restructuring of information if this were the basic representation.

A further problem is that the use of Constructed Objects to describe data items conceals something of their simplicity. The Constructed Object description of something as simple as a list can appear very confusing.

In some ways, this representation provides an appropriate model for tutoring Lisp. For example, the three element list which is constructed by

```
(prefix a (prefix b (prefix c nullist)))
```

is very similar to the method used to construct such a list in Lisp;

```
(CONS A (CONS B (CONS C NIL)))
```

This would allow a direct correspondence to be set up between prefix and CONS. This similarity reflects the common ancestry of the two systems. Both Lisp and Operational Semantics owe much of their underlying form to Church's lambda calculus [Church 1956] . Indeed, one application which was originally suggested for Lisp was its use as a method for comparing programming languages.

The simplicity offered by this similarity would not extend to tutoring other languages than Lisp. For this reason it was a deciding factor in not choosing this form of representation.

In summary, then, operational semantics is capable of simulating the behaviour of a language interpreter. It provides a reasonable model of Lisp for generating descriptions of procedural mechanisms, but is cumbersome when applied to most other languages. It cannot be used to tackle the problems of automatic programming or problem generation.

2.1.2-Axiomatic semantics.

2.1.2.1-General difficulties.

Since the introduction of axiomatic semantics, many papers have been produced showing formal limitations of Hoare logic as a system. A common difficulty is that since the system was designed around program units with a single input and output state, there is a problem with dealing with unusual control flow features. Jumps present problems which have not been adequately dealt with. Certain instances of label can be dealt with by reorganising the loop as a function call [Clint 1972] , and more generally it has been proposed that axiomatic semantics can handle these features if it follows the control structure of a program rather than the textual form. This is returning to the work of Floyd [Floyd 1967] on which axiomatic semantics was based. In this work the inference was applied to a flowchart rather than a program text.

For our purposes, an interesting limitation is that pointed out by M.O'Donnell [O'Donnell 1982] . In this paper it is shown that several rules used in axiomatic semantics are not logically sound, and introduce inconsistencies into the system. He suggests that it is important for a system such as this to "...use a criterion for correctness that corresponds to our intuitive idea of legitimate reasoning.". This is contrasted with certain Hoare logic proofs, in which it is suggested that some rules "...allow intuitively false reasoning which leads by formal tricks to a true result." This has important implications for the "truth" of intermediate results of a proof. It is shown that correct rules can be written, but that they are "unsatisfyingly inelegant". In conclusion, O'Donnell suggests that "convenient and elegant rules for reasoning about certain programming constructs will probably require a more flexible notation than Hoare's."

It is suggested that axiomatic semantics provides an inadequate means of describing user-defined functions. This is not due to a limitation of the mechanism, but to the way in which the results are expressed. It seems intuitively reasonable to want to calculate the effect of a function once, and then to state the result as a lemma which may be directly incorporated into the proof of any program which uses that function. In fact, axiomatic semantics requires the conditions of the function proof of the function to be reestablished each time the function is called.

Having highlighted these problems it should be acknowledged that research into axiomatic techniques is still progressing, and Hoare logic has been applied to some remarkably complex systems (such as parallel processing machines).

2.1.2.2-Application to tutoring.

This semantic system appears to be better suited to providing explanations of programs and reasoning about specific problems than operational semantics. In a complete definition of a programming language [Hoare 1973] , we find fairly simple rules associated with each language statement. While these rules do describe the effect of the statement, they are not in direct correspondence with the descriptions which human programmers would exchange. To derive appropriate descriptions from these representations requires reference to certain global rules used in the language definition, and consequently involves a large amount of computation in organizing this knowledge for presentation.

Apart from the problem of intuitively false reasoning being permitted in Hoare logic, it is unlikely that any of the formal limitations of the system which were mentioned above will cause problems in the task of tutoring novice programmers. Most of these limitations are in areas in which the requirements for correctness of the system can reasonably be relaxed when dealing with novices.

A more serious problem is with the style of proof. While a typical axiomatic semantic proof of the properties of a program would be quite acceptable to an individual with a mathematical background, many others would find it extremely unhelpful. For these people it is a transfer from one unclear formalism (the programming language) to another. The problem is not easily overcome, since it is to do with overall approach rather than some detail such as how the proof is described.

It is apparent that mathematicians generally have some model of the form that a proof will take before attempting to write it rigorously. We must try and access the information which precedes the formal proof if we are to provide psychologically reasonable explanations of programs.

2.1.3-Denotational semantics.

2.1.3.1-General difficulties.

Denotational semantics has proved to be a powerful method for describing program behaviour. Its limits as a representation system have not been fully explored, although there seems to be some difficulty in describing certain classes of parallel and nondeterministic computation. These limits need not concern us here.

2.1.3.2-Application to tutoring.

The denotational description of the execution of a program corresponds quite well to an intuitive model of the way in which an interpreter operates. The description may be manipulated mathematically to deal with other issues in program representation. A possible limitation is that the system is essentially procedural in nature, which may lead to difficulties when attempting to tutor declarative languages. Another source of uncertainty for the pupil is the primacy of recursive representations in Denotational Semantics. Recursion does not appear to be an obvious form of activity to novice programmers.

2.2-Semantic representation used in this system.

2.2.1-Factors influencing choice of representation.

As has been shown in the previous section, none of the existing formalisms for describing the semantics of programming languages is perfectly suited to all the tasks required for tutoring. The design goals of the representation which is actually used focus upon some of these difficulties. These goals will now be outlined.

1) The task of explaining the execution of a program requires a procedural model of the interpreter. This need not be a model of the actual implementation provided that it does not conflict with the behaviour of the interpreter. In fact, there may be a large number of acceptable models, and different models may be suitable for teaching the same concepts to different individuals, or different concepts to the same individual.

For example, consider the following three models of recursion.

i) When a function calls itself, the current value of each local variable is placed on a push-down stack and new values are calculated. When that invocation exits, the old values are retrieved from the stack in the reverse order to that in which they were put on.

ii) Each local variable is a push-down stack. When a new invocation of the function is made, the new value calculated for a given variable becomes the top item on that variable-stack. The next item down is the previous value for that variable, and so on. Each time an invocation of a function is exited, the top item on each variable-stack is thrown away.

iii) When a call to a function is encountered, the system finds the body of the function and substitutes it for the function call, replacing each instance of a bound variable with the appropriate calculation of an argument.

The first two examples are completely acceptable models of recursion. The first is closer to an actual implementation, while the second achieves simplicity by ignoring the available ordering information and positing an arbitrary number of stacks. Since the latter does not require a particular evaluation sequence for arguments it is closer to the mathematically pure idea of a functional architecture. The third example provides an adequate model for purely functional systems, but is wrong in its predictions for certain cases involving side-effects. For example,

```
(DE FRED (X)
       (COND ((ZEROP X) NIL)
             (T (PROGN (SETQ X (ADD1 X))
                       (FRED (SUB1 X)) ))))
```

will recurse indefinitely. If we used the model in iii) to substitute a calculated value for X in a second invocation etc, we would get;

```
(DE FRED (X)
       (COND ((ZEROP X) NIL)
             (T (PROGN (SETQ X (ADD1 X))
                       (COND ((ZEROP (SUB1 X)) NIL)
                             (T (PROGN (SETQ X (ADD1 (SUB1 X)))
                                       (FRED (SUB1(SUB1 X))) ))))
```

we can see that the value of X actually decreases for the next substitution, so this model predicts that the function will terminate.

Despite its problems, the third model is very useful. It can be shown as a simple extension of more general function invocation, and will behave perfectly in most situations.

Each of the formal descriptions given earlier (with the exception of Denotational semantics) implies one particular procedural model of the language. A goal of the chosen formalism was to support an explanatory mechanism which could offer different procedural models of the same basic representation.

2) When a real programmer or student describes the action of a program, they often talk in terms of the "state" of the execution at a particular time, or the "properties" of a particular statement. Empirical studies have shown [Sime 1973] that programmers tend to analyse programs in terms of "states" as opposed to procedural features such as control flow. These declarative statements about a program are a necessary part of abstracting away from the properties of a particular machine. The semantic representation is intended to support these static descriptions of a program.

3) The primitives used by each of the systems mentioned above are chosen for their expressive power in a formal mathematical sense. This is not necessarily equivalent to their expressive power as concepts for producing reasonable descriptions for human beings to use. The primitives must often be explained before they can be used. A good example of this is the use of "dot notation" for representing lists. This notation is a basic unit in all the systems which have been described and corresponds to a typical method of implementing list representations in a computer. It divides a list into the complementary pair of "first element" and "everything except the first element". A human being (unlike a computer) may examine a list starting from either end, so this unbalanced representation of a list is far less flexible than that used by humans. I would suggest that humans are able to use "the nth element", "the elements before n", and "the elements following n", as reasonable primitives of list manipulation.

In choosing the primitives of this system, the process of explaining them to the pupil has been borne in mind, so it is hoped that they are more immediately understandable than the basic concepts of other systems.

2.2.1.1-Limitations of this approach.

Before describing the semantics in detail, it is best to acknowledge certain limitations of the representation which is used here. These may not be absolute limits of this form of representation, but are aspects which were too complex to explore in the available time, or which were not of fundamental importance to this application.

1) The representation is a form of Direct Semantics. This means that the state resulting from executing one piece of code is always passed directly to the textually adjacent section of code. Such an approach leads to a simpler semantics for individual statements than would be the case with a Continuation semantics, but unusual control flow features - such as jumps and exits due to errors - cannot be adequately represented. Some special cases can be handled by unusual techniques (e.g. [M.Clint 1972]), but a consistent general method for dealing with these features cannot be provided. The semantic representation used in IMPART assumes that jumps do not exist, and deals with errors by patching a temporary solution and continuing the execution.

2) Representing variables so that their behaviour is correctly modelled under all circumstances is difficult. In particular, a complete modelling of aliasing and shared binding is hard to produce. Modelling the difference between call by value and call by reference languages also requires a flexibility which has not been explored in this representation. LISP, in particular, treats variables in ways which are difficult to model.

Another problem of the system used here is that it is difficult to represent holes in variable scoping. In particular all local variables are effectively scoped as fluids. This limitation should be reasonably straightforward to overcome, but no attempt to do so has been made.

3) Many formal program description techniques focus on the difficult problem of proving whether a program will terminate or not. The present system does not consider this problem. Expressions are executed without proving that they will terminate.

4) Effort has not been expended on achieving mathematical purity for this representation. It is not a mathematically minimal system, nor is it sufficiently complete or consistent to model all formal languages. The representation is probably more powerful than it need be to deal with context-free languages. It is hoped that work on this area can be continued.

5) The semantics of datatypes have not been dealt with in sufficient degree. In particular, lazy typing can cause problems. For example, NIL may be regarded as an atom, a list or a boolean in Lisp. If these roles are mixed (e.g. by CONSing onto the value of a failed predicate), then the current version of the semantics will regard it as illegal.

Another problem is with user-defined datatypes. Although simple datatypes have been explored, the system is not in a state which is capable of dealing with embedded datatypes. For example, a PASCAL record whose fields are themselves records cannot be handled.

3.-Summary of chapter 9.

Five major problems which must be tackled in the tutoring system have been described. Each of them makes use of knowledge about the semantics of the programming language which is being tutored. Traditional methods of representing program semantics have been summarized, and their application to tutoring considered. The reasons for opting for a new formalism have been given and the limitations of that formalism have been discussed.

Chapter 10.

Using a semantic representation for tutoring.

1.-Primitive elements of the representation.

The representation used is essentially a form of predicate calculus. In fact, the semantics are implemented in PROLOG to simplify the task of mechanization. For this reason they will be shown in PROLOG syntax. No knowledge of the language itself is assumed.

1.1-Communication.

The basic model for communication involves the existence of an arbitrary number of named "channels" which link program statements together. Statements communicate using the primitives IN and OUT. Each of these primitives has two arguments. In both cases the first argument is a channel name. These names must match if information is to pass from an OUT statement to an IN statement. The second argument to OUT may be any piece of data. The second argument to IN must be a variable. The effect is to bind that data to occurences of that name. For example,

```
out($value$,4),out($printer$,hello),
        ...,
in($value$,VAL),in($printer$,PR)
```

has the effect of replacing VAL with 4 and PR with hello. It should be noted that the naming of channels permits the representation of the semantics of parallel communicating processes. The "value" channel is commonly used to return the values of expressions.

1.2-Control flow.

In order to determine the effect of a particular statement it is often necessary to determine the effect of subcomponents within that statement. For example, the value of (ADD1 X) depends on the value of X. The DO primitive takes one argument and represents determining the effect of that argument. If I wish to determine the behaviour of "add1(3)" I would represent this as "do(add1(3))", and somewhere within the description of add1(3) will be the statement "do(3)".

It is worth noting that allowing the DO primitive to appear in statement descriptions permits the recursive definition of statements. The representation does not have an iterative control flow construct.

Some language statements must represent a choice between two or more possibilities; (this is the case with conditional statements, for example). A choice is represented using the OR predicate. This predicate may have two or three arguments, each of which is a list of semantic primitives. As an example, "IF test THEN action1 ELSE action2" would include in its description the primitive;

```
or([test,action1],[action2]).
```

The way OR operates is to start with the leftmost argument and carry out every item in that list. If it reaches the end of the list then it has succeeded, otherwise it repeats the process for the second argument. An error situation arises if OR cannot get to the end of any of the lists.

Another useful predicate for control flow is DONOTHING. This, as one would expect, does nothing. The predicate is used in conjunction with OR when one path requires no action to be taken (e.g in a while-do statement).

The EQUALL predicate is a test which takes two arguments. If the arguments are identical then the test succeeds, otherwise it fails.

1.3-Side-effects.

Side-effects are represented by introducing the concept of an "environment" which holds information about the value of variables, scope of identifiers and so on. Special items like scoping are represented by "tags" which are identifiers surrounded by "$" signs. There are four primitives which can manipulate the environment;

1) CREATE - This primitive takes two arguments, the first being an identifier and the second being an item associated with that identifier. It adds this pair as the most recent thing in the environment. It does not affect any existing members of the environment. Examples of using this primitive would be

```
create(deal,5)
create(deal,$local$)
```

which may be taken to mean assign value 5 to "deal", and declare "deal" as a local variable, respectively. The environment maintains a recency ordering, so that

```
create(deal,5)
create(deal,8)
```

may be used to represent a variable which currently has value 8, but which will revert to a value 5 when the program exits from the current scope declaration.

2) DELETE - This takes two arguments. The first is an identifier and the second matches an item in the environment. The primitive removes the most recent member of the environment which matches its arguments. If no such member is found, it fails. The most common use of this and the previous predicate is to change variable declarations. The effect of a PROG statement which declares X local will include the primitives

```
create(x,nil)
create(x,$local$)
```

near the beginning and

```
delete(x,_)
delete(x,$local$)
```

near the end.

3) CHANGE - This has two arguments. It finds an expression whose left hand side is equal to the first argument and modifies the right-hand side of that environment pair to be equivalent to the second argument. This primitive is not strictly necessary since the task could be achieved by a DELETE/CREATE pair, but these pairs occur with such frequency that it is convenient to have a more concise version.

```
before: [[x,5],[y,7],[x,fred]]
    change(x,different)
after: [[x,different],[y,7],[x,fred]]
```

4) SEE - This takes two arguments. The first being an identifier and the second being an item. It succeeds if it can find a corresponding pair in the environment, and fails otherwise. As an example,

```
see(x,$local$)
```

is a means of checking whether x has been declared as a local variable.

1.4-Primitive datatypes.

There are three datatypes recognised by the semantic system;

1) Numbers - The semantics are capable of dealing with positive and negative integers. The operations PLUS, TIMES, DIFFERENCE, DIVIDE and GREATER are defined on these numbers.

2) Identifiers - A usable identifier is taken to be a string of alphanumeric characters. There are no special operations defined on identifiers.

3) Ordered sets - Every other object is an ordered set of elements (which may themselves be ordered sets). Each set has a set-type associated with it (such as **list). These have no effect on the actions which may be performed on the set, but indicate a particular type of object in the target language with which that set may be associated. For example,

```
**list(a,b,c)
**vector(a,b,c)
```

are two different datatypes in the target language (one is a list, the other a vector), but are effectively indistinguishable in terms of the primitive operations which may be carried out on them. There is one special case of this primitive, which is the empty set. There are six operations defined on ordered sets;

i) ELEMENT(N,SET,NAME) - gives name NAME to the Nth element of SET.

```
ELEMENT(3,*(a,b,c,d),c)
```

ii) BEFORE(N,SET,NAME) - NAME becomes a set of those elements which precede the Nth element in SET.

```
BEFORE(3,*(a,b,c,d),*(a,b))
```

iii) AFTER(N,SET,NAME) - NAME becomes a set of those elements which follow the Nth element in SET.

```
AFTER(3,*(a,b,c,d),*(d))
```

iv) ADDELEMENT(N,ITEM,SET,NAME) - NAME is a new set, like SET but with an additional element ITEM in the Nth position.

ADDELEMENT(3,zz,*(a,b,c,d),*(a,b,zz,c,d))

v) ADDBEFORE(N,NEWSET,SET,NAME) - NAME becomes a set, like SET but with the elements of NEWSET added before the Nth position.

ADDBEFORE(3,*(x,y,z),*(a,b,c,d),*(a,b,x,y,z,c,d))

vi) ADDAFTER(N,NEWSET,SET,NAME) - Like ADDBEFORE, but the new elements appear after the Nth position.

ADDAFTER(3,*(x,y,z),*(a,b,c,d),*(a,b,c,x,y,z,d))

1.5-Target language datatypes.

As part of the description of a language, it is necessary to provide a description of all the basic types of object which are used in the language. This must cover types of program statement as well as data types. In this formalism object-types are defined by a series of "type" predicates which describe the types of the target language in terms of the primitive datatypes of the semantic representation. Some work has been done on formalizing the semantics of datatypes [Hoare 1972] but since the types of object in a given language are very diverse and have few properties which are shared due to underlying principles, this system has made no attempt to generate a set of semantic primitives which will describe all datatypes. Types are described by Prolog clauses such as those shown below. Since this is a bottom-level of the system "understanding", methods for talking about these datatypes must also be provided with a particular language.

```
            sexpr
         .    .    .
       .        .     .
     atom     list     Vector
   . . .      .  .
  .  .  .     .     .
```

num bool alpha biglist dotted-pair

Figure 13 - Lisp datatypes.

This figure illustrates the hierarchy of datatypes in LISP. Each of these datatypes must be described in terms of the primitive datatypes. For example, the number 3 is an instance of "num". More generally it may be regarded as a special case of an "atom", and as one particular type of "sexpr".

The following specification is a complete set of type descriptions for LISP. The general convention is that the first argument corresponds to a particular object whose type we wish to determine, while the second gives the name of that type of object. In the instance of "t", for example, rule (2) simply indicates that "t" is a boolean. The more complex datatypes include a right-hand side (following the ":-" operator), which express the constraints upon a particular sort of datatype in terms of the primitives of the semantic representation. An example of this is rule (4), which states that any object ?X is an instance of "alpha" if it corresponds to an identifier in the semantic representation. It will be recalled that there are three types of primitive to which datatypes may be related; identifiers, integers and ordered sets. These are identified by the corresponding predicates "prim_identifier", "prim_integer" and "prim_o_set".

It was mentioned above that an item may belong to several categories of object-type. In the specification shown below the rules are ordered; rule (1) is the most specific rule and rule (8) is the least specific. It can be seen that an integer could be identified as a number by rule (1), as an atom by rule (5) and as an sexpr by rule (8).

```
1) type(?X,number):-prim_integer(?X).
/* Anything is a number if it is a primitive integer */

2) type(t,boolean).
/* t is a boolean  */
```

```
3) type(nil,boolean).
/* nil is a boolean  */

4) type(?X,alpha):-prim_identifier(?X).
/* Anything is an alpha if it is a primitive identifier  */

5) type(?X,atom):-type(?X,number);
                  type(nil,boolean);
                  type(?X,alpha).
/* Anything is an atom if it is a number or a boolean or an alpha  */

6) type(?X,list):-prim_o_set(?X),
                    o_set_name(?X,**list).
/* Anything is a list if it is a primitive ordered set called "**list"
*/

7) type(?X,vector):-prim_o_set(?X),
                    o_set_name(?X,**vector).
/* Anything is a vector if it is a primitive ordered set called
"**vector"  */

8) type(?X,sexpr):-type(?X,atom);
                   type(?X,list);
                   type(?X,vector).
/* Anything is an sexpr if it is an atom or a list or a vector  */
```

One further example will illustrate the application of this technique to giving types to particular language expressions. This predicate defines an expression as any statement which returns a value;

```
type(X,expression):-
        effect(X,PRE,POST),
        member(out($value$,_),POST).
```

1.6-User defined datatypes.

To illustrate the operation of user-defined datatypes, let us consider defining the following record in PASCAL;

```
VAR fred:RECORD  x,y:num;
```

This involves creating a variable "fred" which has two numbers associated with it that may be referenced as fred.x and fred.y respectively. This is achieved by defining "." as a binary operator in PASCAL, which searches the environment for an instance of its first argument with the second argument in its tag-field. Essentially our definition of "fred" will add

```
[[fred,$fluid$],[fred,[x,*unbound]],[fred,[y,*unbound]]]
```

to the environment. This will be deleted when the scope of fred is left.

1.7-Describing the effect of an individual statement.

For each statement in the language, a description is produced which lists the preconditions for carrying out that statement and then gives a sequence of actions which correspond to the effect of the body of the statement. Further examples of statement descriptions can be found in appendix 2, where a reasonably complete description of LISP is given.

As a simple example, consider representing the behaviour of a number in a programming language. The effect of executing a number, such as 1, is to return that number as the value of the execution. This is represented as follows;

```
effect(1,
        [],
        [out('$value$',1)]).
```

This representation is called an "effect" predicate. It should be regarded as an object "effect" which corresponds to a description of the action of a language statement. The predicate has three arguments which are shown on separate lines, delimited by commas. A full stop indicates the end of the description. In this case, argument 1 is the number 1. This simply identifies the statement which this effect applies to. The second argument is the empty list []. This fills the position at which we would expect to find preconditions of the statement. In this case it indicates that when we are dealing with the number 1, no further tests need be made. The final argument represents the body of the function. In this case only one action is present. This is an OUT action which, as explained above, has the net effect of causing the expression to return the value 1 as its result.

Clearly, it is not possible to provide such an effect description for each member of the infinite set of numbers. Instead, an "effect schema" is defined. This is an effect predicate similar to that given above, but which may match a number of program statements. If the schema successfully matches a particular statement, then it becomes "instantiated", to produce an effect description for that particular statement. In the following example we have a schema for numbers. The particular instance of the number 1 would match this and it would become instantiated to the same "effect" description as that given above.

```
effect(NUM,
        [],
        [out('$value$',NUM)])
      :-type(NUM,number).
```

Some additional features of the representation have been introduced in this example. "NUM" is a variable. It may match any statement but will become instantiated to a particular number if the schema is applied to a number. It is a convention that all variables will be shown in capital letters. It will also be noticed that this schema has a right-hand side (following the :- sign). This provides constraints upon the language statements which may match a particular schema. In this case it indicates that this schema will only apply to statements which are numbers. It is important to note the distinction between instantiating a schema to produce an effect description, and instantiating an effect description to produce a complete description of the action of a statement.

It should be observed that the right-hand side of the schema does not serve the same purpose as the preconditions of the "effect" (argument 2). The preconditions are a feature of a particular effect description, while the right-hand side is part of the instantiation mechanism which produces that effect description from a schema. It should be apparent that the following schema is incorrect since it confuses these roles.

```
effect(NUM,
        [type(NUM,number)],
        [out('$value$',NUM)]).
```

Let us now consider the application of these descriptions to the lisp function ADD1. This function expects one argument which is a number, and returns a value one greater than that argument.

```
effect(add1(ARG1),              /* 1 */

       [type(ARG1,sexpr)],      /* 2 */

       [do(ARG1),               /* 3 */
        in('$value$',VARG1),
        type(VARG1,number),
        plus(1,VARG1,RES),
        out('$value$',RES)]).
```

The first argument /* 1 */ is more complex than in the previous example. It provides an "abstract syntax" of the function ADD1 which indicates that it has one argument. Ordering of arguments is the same as that of the language being described, but no other "syntactic spice" is preserved. In this case, as with most statements, the abstract syntax is sufficient to determine whether a given expression can be represented by this schema. Use of a right-hand side is rare. If we attempt to describe the general properties of ADD1 using this schema, we would have the following description;

> " ADD1 takes one argument {add1(ARG1)}. The argument must be an s-expression (any lisp expression) {type(ARG1,sexpr)}. The argument is evaluated {do(ARG1)} and it returns a value {in('$value$',VARG1)}. This value must be a number {type(VARG1,number)}. One is added to the number {plus(1,VARG1,RES)} and this is returned as the value of the function {out('$value$',RES)}."

Note that this schema will match any instance of ADD1 irrespective of the type of argument which it has. For example, (ADD1 (QUOTE X)) will produce the legitimate instantiation;

```
effect(add1(quote(x)),
       [type(quote(x),sexpr)],
       [do(quote(x)),
        in('$value$',VARG1),
        type(VARG1,number),
        plus(1,VARG1,RES),
        out('$value$',RES)]).
```

The system detects the problem with this expression if it actually attempts to fill in the undetermined parts of the instantiation. The step "do(quote(x))" will place a value X in the environment which will be substituted for all instances of VARG1. The body will therefore contain the test "type(x,number)" which will fail. Any item in the preconditions or body which fails would correspond to an error being generated by a normal interpreter. By contrast, here is an example of the description of (ADD1 (QUOTE 3)) after it has been successfully carried out;

```
effect(add1(quote(3)),
       [type(quote(3),sexpr)],
       [do(quote(3)),
        in('$value$',3),
        type(3,number),
        plus(1,3,4),
        out('$value$',4)]).
```

Consider the more complex example of a "while_do_" statement in PASCAL. This statement executes its second argument until the test which is represented by its first argument fails. It then terminates. The choice of paths in a statement is represented by the predicate "or". An "error" will only be generated inside an "or" if all the alternative paths fail.

```
effect('while_do'(ARG1,ARG2),
        [type(ARG1,expression),
         type(ARG2,statement)],
        [do(ARG1),
         in('$value$',VARG1),
         type(VARG1,boolean),
         or([equall(VARG1,true),
              do(ARG2),
              do('while_do'(ARG1,ARG2))],
            [equall(VARG1,false),
             donothing])]).
```

In the "or" statement, the first path corresponds to the case where the test (of the while statement) is true. In this case the second argument is executed and the "while_do" schema is used recursively. If the test fails, the second path of the "or" is followed and no further action is taken.

1.8-Representing user-defined functions.

The representation of pieces of code that have been named by the user (such as LISP function definitions or PASCAL procedure calls) involves adding an effect description to the language which will recognise a user-defined function, retrieve the appropriate code, and channel the appropriate data to and from that function. If such an effect descriptor exists, then it is only necessary to place appropriate code in the environment when a function is defined. Consider defining a function KATHLEEN;

```
(DE KATHLEEN (A B)
```

```
(LIST A (ADD1 B)))
```

The description of DE in appendix 2 shows that the net effect of this is to add

```
[[kathleen,[$expr$,[a,b],list(a,(add1(b)))]]
```

to the environment. Invoking KATHLEEN will match the description

```
effect(X(ARG1,ARG2),
        [see(X,[$expr$,[PAR1,PAR2],BODY])],
        [do(ARG1),
         in($value$,PAR1),
         do(ARG2),
         in($value$,PAR2),
         do(BODY),
         in($value$,VBODY),
         out($value$,VBODY)]).
```

In this case, matching the function call with the environment has the effect of retrieving the parameters and body. Each argument is executed, and the resulting value bound to the appropriate parameter. The body is then executed, with the value being passed back to the statement which invoked this function.

2.-Formal descriptions of toy domain.

This section illustrates the foregoing description by providing a more formal outline of the way in which the semantics of the language used in the toy domain of chapter 7 is represented.

The formal semantic descriptions of this toy language will now be given to show the practical aspects of this representation. They are presented in the form of predicates written in the language PROLOG. These predicates are manipulated by a variety of programs in order to carry out tasks such as descriptive output, execution of commands, and planning a command sequence. An equivalent definition for LISP can be found in appendix 2.

This is the description of the action PUSH. It should be compared with the English description in figure 8.

```
effect(push(ARG1,ARG2), (1)
        [type(ARG1,actor),    (2)
          type(ARG2,pushable)],  (3)

        [or([ see(bolt,$down$)],  (4)
            [ see(ARG2,$open$),
              change(ARG2,$shut$)],
            [ see(ARG2,$shut$),
              change(ARG2,$open$)] )]
                                             ).
```

Figure 14 - Semantics of PUSH.

Line 1 will match with a particular example of the action. For instance, MARY PUSH DOOR would match this effect by replacing all occurences of ARG1 with MARY, and all occurences of ARG2 with DOOR. Notice that all actions are reduced to this prefix format which retains the original argument ordering, but ignores syntactic "spice".

Lines 2 and 3 together constitute the preconditions for the application of the function. In line 2 a check is made to ensure that the actor is a possible actor in this world. Line 3 makes sure that the object being pushed is a member of the class of pushable objects, although in this case the sparseness of the world means that the only member of that class is the door.

Line 4 to the end is the body of the action. In this case it consists of a single "or" statement, whose arguments represent 3 alternative possible ways to manipulate the world. Each argument is a list of statements consisting of tests on the environment or changes to be made in the environment. The interpreter will report a semantic error in a command if none of these alternatives can be completely carried out. The statements used in this case are "see", which tests the presence of a fact in the environment, and "change" which replaces a fact with another one.

For this particular action, it is not possible to violate the semantics in the body, because it is defined in such a way that if it is not possible to move something then the function completes successfully but has no effect. Actions are not generally so robust. If it is preferable for an error to result from attempting to push an object when something is preventing it from moving, the definition would be rewritten as follows;

```
effect(push(ARG1,ARG2),
        [type(ARG1,actor),
          type(ARG2,pushable)],

        [or([ see(bolt,$up$),  /* Changes here */
              see(ARG2,$open$),
              change(ARG2,$shut$)],
            [ see(bolt,$up$),
              see(ARG2,$shut$),
              change(ARG2,$open$)] )]
                                                  ).
```

Figure 15 - Extended semantics of PUSH.

Sliding is an action which can be done to any slidable object (only the bolt in this case). It may be done by either actor, so long as they are inside. It results in the bolt becoming up if it was down, or down if it was up. This would be expressed in the following way;

```
effect(slide(ARG1,ARG2),
       [type(ARG1,actor),
        see([ARG1,$in$]),
        type(ARG2,slidable)],

       [or([ see(ARG2,$up$),
             change(ARG2,$down$)],
           [ see(ARG2,$down$),
             change(ARG2,$up$)])]
                                        ).
```

Figure 16 - Semantics of SLIDE.

An example of a violation of this semantic description would be " cannot satisfy see([john,in]) in the preconditions of slide" i.e. JOHN cannot slide the bolt because he is not in.

Move has the special requirement that it's actor and object are the same. It only has an effect if the door is open, when it moves actors out if they are in, or in if they are out;

```
effect(move(ARG1,ARG2),
       [type(ARG1,actor),
        equall(ARG1,ARG2)],

       [or([ see(door,$shut$)],
           [ see(ARG1,$in$),
             change(ARG1,$out$)],
           [ see(ARG1,$out$),
             change(ARG1,$in$)])]
                                        ).
```

Figure 17 - Semantics of MOVE.

In the above definitions, all predicates (such as "see") are primitives which are defined within the "effect interpreter" with the exception of "type", which identifies the basic datatypes of the language. This predicate must be provided as part of the language description. It takes two arguments, the second being the name of the type of the first. These are written as normal prolog clauses. For this world they are as follows;

```
type(john,actor).
type(mary,actor).
type(door,pushable).
type(bolt,slidable).
type(_,object). /* i.e. Everything is an object */
```

As an example of the way in which this system is extensible, consider adding an extra actor which will behave exactly like the others. This simply involves adding the type description;

```
type(suzi,actor).
```

If an extra object which behaves differently is added, then the actions must be rewritten accordingly. For example, adding a WINDOW which may be PUSHed and is independant of the state of the BOLT would require PUSH to be modified as follows;

```
effect(push(ARG1,ARG2),    (1)
       [type(ARG1,actor),   (2)
        type(ARG2,pushable)],   (3)

       [or([ equall(ARG2,door),see(bolt,$down$)],   (4)
           [ see(ARG2,$open$),
```

```
                change(ARG2,$shut$)],
              [ see(ARG2,$shut$),
                change(ARG2,$open$)] )]
                                                        ).
```

Figure 18 - Alternative semantics of PUSH.

i.e. one extra test has been added to line 4. It is also necessary to add a type predicate which extends the set of pushable objects;

```
type(window,pushable).
```

3.-Relationship to other formal semantic systems.

The formalism used here may be compared with other methods of describing programming language semantics. It may be viewed in a variety of ways, depending upon the manner in which it is applied.

The primitives of the representation may be regarded as primitives of a simple machine, as with Operational semantics. This is not the most productive way to regard the system, since it precludes the sort of reasoning about the semantics which will be described in the next section. Although the effect descriptions may be read declaratively as assertions about the state of the environment before and after a statement is executed, it is not immediately clear whether these descriptions provide a basis for linking assertions about relationships between states as is done in Axiomatic semantics. It may be possible to regard the effect descriptions as purely functional mappings between states, as in denotational semantics, but this requires further discussion (see below).

Other differences of this representation are the choice of primitives (particularly those for list processing), and the fact that the system allows names to be assigned to individual objects within a program, rather than to groupings of objects. It may also be noted that recursively defined statements are of great importance to this style of language definition.

3.1-Equivalence with denotational semantics.

Since denotational semantics is a very powerful representation system about which many results have been proved, it would be extremely useful to demonstrate an equivalence between this system and the Prolog representation given above. As it stands the Prolog system has acknowledged deficiencies and any formal attempt to prove an equivalence would undoubtedly fail. Instead an informal comparison will be provided with the proviso that future extensions to this representation will be made with reference to ways of improving this correspondence with denotational semantics. This section briefly outlines the basic ideas of denotational semantics, gives two definitions of the language TINY [Gordon 1979] (for comparison), and concludes by stating some rules for transforming between the Prolog representation and denotational semantics.

3.1.1-Denotational semantics.

Denotational semantics is the most complete descriptive formalism for programming languages which has been developed up to the present. It combines a complete and elegant mathematical structure with an intuitively reasonable descriptive theory.

Only a direct denotational semantics will be outlined. A direct semantics is a system in which the result of executing one expression is passed to the textually adjacent expression in the program (similarly to axiomatic semantics). A more complex form of semantics which is capable of handling awkward control flow features, is a Continuation semantics. Here the idea of a CONTINUATION associated with each expression is used to formalise the possibility of execution moving to an arbitrary point in the program text.

Continuation semantics will not be further discussed.

3.1.1.1-Describing statements.

The overall approach is to associate each program statement with a DENOTATION - an abstract mathematical entity which models the meaning of that statement. These denotations are described by SEMANTIC FUNCTIONS which are mathematical functions providing mappings between STATES, VALUES and ERRORS. The basic form is an expression

P[P']s

which represents P, the denotation of P' with respect to state s.

A VALUE is defined to be a member of the set of denotable values defined for a particular language (this may contain integers, identifiers, complex numbers etc.). Let us also allow {error} to be the set of possible error conditions for a given language. A STATE consists of 3 components;

MEMORY - a correspondence between identifiers and values,
INPUT - list of all inputs to a program
OUTPUT - list of all outputs from a program

Language statements may be divided into Expressions and Commands, the distinction being that the former return a value. This difference is reflected in the range of their denotations;

Semantic function of expressions;

```
E[E]s   E:EXP-->[State-->[[Value*State]+{error}]]
```

E is the denotation of an expression E with respect to state s. The denotation takes a state, and either maps it onto a state and a value, or onto an error.

Semantic function of commands;

C[C]s   C:[State-->[State+{error}]

C is the denotation of a command C with respect to state s. The denotation is a mapping which starts from a state and either maps onto a state or an error.

Let us represent our environment as a value-state pair, where the state is a triple of memory-input-output (i.e. (v,s) and (v,(m,i,o)) represent environments). The following examples give denotations for some simple expressions.

E[1]s = (1,s)   {1}

1 is an expression which has a value of 1 and leaves the current state unchanged.

E[I](m,i,o) = (mI = unbound)->error,(mI,(m,i,o))   {2}

In this case, I is any identifier. This denotation expresses the fact that if I is associated with the special value "unbound" in memory, then executing this expression results in an error. In other cases, the expression returns a value which is the item associated with I in memory, and leaves the original state unchanged. This description refers to the function m, which is a mathematical function associating variable names with values.

C[I:=E]s =(E[E]s=(v,(m,i,o)))->(m[v/I],i,o),error   {3}

The assignment statement is a command. It determines the denotation of the right hand side of the assignment, and calls the resulting state (v,(m,i,o)). If this succeeds then it returns a state in which the memory has been modified to give the value v to I, otherwise it returns an error. This illustrates a further use of the memory function in which m[v/I] is taken to mean: modify the memory such that I has value v.

3.1.2-Comparative definitions of TINY.

In order to demonstrate the equivalence of this system to denotational semantics let us consider two definitions of the semantics of a simple language [M.J.C.Gordon 1979] called TINY. The aim is to give an intuitive feel for the correspondence between the definitions. The denotational form chosen is slightly simpler than the "standard" representation.

The syntax of TINY can be given in a Backus-Naur form as follows;

```
E::= 0|1|
     true|false|
     read|
     I|
     not E|
     E1=E2|
     E1+E2

C::= I:=E|
     output E|
     if E then C1 else C2|
     while E do C|
     C1;C2
```

where E represents expressions and C represents commands.

For the purposes of the denotational description, we will define the following domains

Ide={I|I is an identifier}

Exp={E|E is an expression}

Com={C|C is a command}

and formalise the state as follows;

State=Memory x Input x Output

Memory=Ide --> [Value+{unbound}]

Input=Value*

Output=Value*

Value=Num + Bool

Where * indicates a string (sequence of the items preceding the star).

The formalized state corresponds to a "batch process" model of input and output where all inputs are present in the environment before execution and all outputs are present in the environment after execution. In the prolog formalism this will be modelled by the use of special tags "$input$" and "$output$". We will now give denotational and prolog forms of the semantics of each statement.

(E1) E[0]s = (0,s)

E[1]s = (1,s)

(E1*)
```
effect(0,
       [],
       [out('$value$',0)]).

effect(1,
       [],
       [out('$value$',1)]).
```

(E2) E[true]s = (true,s)

E[false]s = (false,s)

(E2*)
```
effect(true,
       [],
       [out('$value$',true)]).
```

```
effect(false,
        [],
        [out('$value$',false)]).
```

These statements are clearly equivalent. The prolog version seems more like an "action", but it may be read in a declarative form. The use of replace with tags is more complex than setting the value of a denotational state, but it provides a more powerful mechanism which indicates the generality of the prolog "environment".

```
(E3)    E[read](m,i,o) = null i
                             ->error,
                               (hd i,(m,tl i,o))
```

```
(E3*)   effect(read,
               [],
               [in('$input$',VAL),
                out('$value$',VAL)]).
```

E3 introduces the use of an if-then-else construct to allow for the possibility of an error in the execution of this statement. An equivalent item is not needed in E3* because the possibility of error is implicitly present in all the descriptions. The denotational description uses "HD" and "TL" to modify the input string, making some reference to the idea of a sequence of such objects. The Prolog form refers only to the item directly involved. The overall effect upon the "state" is not depicted as clearly in the Prolog representation as in E3.

```
(E4)    E[I](m,i,o) = (mI = unbound)
                           ->error
                             ,(mI,(m,i,o))
```

```
(E4*)   effect(ATOM,
              [],
```

```
            [see([ATOM,['$value$',VAL]]),
             out('$value$',VAL)]).
```

In E4, mI corresponds to finding the value associated with I in memory. This is exactly equivalent to the "find" statement in E4*. The final "out" has no net effect on the state, and may be considered confusing when compared with the obviously unchanged state of E4.

```
(E5)   E[not E]s = (E[E]s=(v,s'))
                     ->isBool v
                          ->(not v,s')
                            ,error),
                       ,error


(E5*)  effect(not(ARG1),
            [type(ARG1,expression)],
            [do(ARG1),
             in('$value$',VARG1),
             type(VARG1,boolean),
             or([equall(VAL,false),
                 out('$value$',true)],
                [equall(VAL,true),
                 out('$value$',false)])]).
```

E5* explicitly represents the fact that ARG1 must be an expression. The "do" predicate corresponds to determining the denotation of the subexpression E in E5. The alternative possibilities are more clearly visible in E5*, where the use of an "or" predicate which permits an arbitrary number of possibilities subsumes the role of several if-then-else expressions. The "isBool" primitive is equivalent to a "type" predicate. E5* explicitly manipulates the truth values whereas E5 refers to a primitive boolean operation to achieve this. The lack of explicit mention of error states simplifiies the description in E5*. Both formalisms have introduced extra symbols to name intermediate states in the process of execution.

```
(E6)   E[E1=E2]s = (E[E1]s=(v1,s1))
                         ->((E[E2]s1=(v2,s2))
                             ->(v1=v2,s2)
                               ,error)
                           ,error

(E6*) effect(equal(ARG1,ARG2),
            [type(ARG1,expression),
             type(ARG2,expression)],
            [do(ARG1),
             in('$value$',VARG1),
             do(ARG2),
             in('$value$',VARG2),
             or([equall(VARG1,VARG2),
                 out('$value$',true)],
                [out('$value$',nil)])]).
```

The only new item introduced here is the explicit test for equality "equall", which is equivalent to a denotational equality primitive.

```
(E7)   E[E1+E2]s = (E[E1]s=(v1,s1))
                             ->((E[E2]s1 = (v2,s2))
                                 ->(isNum v1 and isNum v2
                                     ->(v1+v2,s2)
                                       ,error)
                                   ,error)
                               ,error

(E7*) effect('+'(ARG1,ARG2),
            [type(ARG1,expression),
             type(ARG2,expression)],
            [do(ARG1),
```

```
              in('$value$',VARG1),
              type(VARG1,number),
              do(ARG2),
              in('$value$',VARG2),
              type(VARG2,number),
              plus(VARG1,VARG2,RES),
              out('$value$',RES)]).
```

The denotational description introduces isNum (equivalent to a "type" predicate) and an addition primitive. A similar addition primitive is introduced in E7*.

```
(C1)   C[I:=E]s = (E[E]s=(v,(m,i,o)))
                    ->(m[v/I],i,o)
                       ,error

(C1*) effect(':='(ARG1,ARG2),
             [type(ARG1,atom),
              type(ARG2,expression)],
             [do(ARG2),
              in('$value$',VARG2),
              change(ARG1,['$value$',VARG2])]).
```

Typechecking is explicit in C1*. The function m[v/I] corresponds to a replace statement where the identifier is the first item.

```
(C2)   C[output E]s = (E[E]s = (v,(m,i,o)))
                          ->(m,i,v.o)
                             ,error

(C2*) effect(output(ARG1),
             [type(ARG1,expression)],
             [do(ARG1),
```

```
              in('$value$',VARG1),
              out('$output$',VARG1)]).
```

In C2, the string of outputs is modified by using the "." operator which divides a list into first and rest.

```
 (C3)   C[if E then C1 else C2]s = (E[E]s = (v,s'))
                                              ->(isBool v
                                                  ->(v
                                                      ->C[C1]s'
                                                        ,C[C2]s')
                                                    ,error)
                                                ,error
```

```
 (C3*) effect('if_then_else'(ARG1,ARG2,ARG3),
              [type(ARG1,expression)],
              [do(ARG1),
               in(['$value$',VARG1]),
               or([equall(VARG1,true),
                   do(ARG2)],
                  [equall(VARG1,false),
                   do(ARG3)])]).
```

The use of a value as a test for the "->" construct is introduced in C3. Nothing new is added in C3*.

```
 (C4)   C[while E do C]s = (E[E]s = (v,s'))
                                    ->(isBool v
                                        ->(v
                                             ->((C[C]s' = s'')
                                                  ->C[while E do C]s''
                                                ,error)
                                           ,s'),
                                       ,error
```

```
(C4*) effect('while_do'(ARG1,ARG2),
             [type(ARG1,expression),
              type(ARG2,statement)],
             [do(ARG1),
              in('$value$',VARG1),
              type(VARG1,boolean),
              or([equall(VARG1,true),
                  do(ARG2),
                  do('while_do'(ARG1,ARG2))],
                 [equall(VARG1,false),
                  donothing])]).
```

```
(C5)   C[C1;C2]s = (C[C1]s = error) ->,C[C2](C[C1]s)
```

```
(C5*) effect(';'(ARG1,ARG2),
             [],
             [do(ARG1),
              do(ARG2)]).
```

C5 illustrates the simplicity of the compound statement operator. Note the use of a null "then" part in the if-then-else construct.

3.1.3-Rules of equivalence.

We will provide a set of rules for transforming individual primitives of a direct denotational semantic representation (slightly simpler than the standard representation) to a prolog semantic representation.

A few global comparisons can be made. Firstly it should be noted that denotational semantics represents an expression as a mathematical transformation between states. The Prolog representation may also be regarded in this way, although the states are implicit in the representation rather than being formalised. Where denotational semantics names separate states this formalism names individual items within a state - the set of these named items at any time constitutes a state.

Another point of comparison is error-situations. Errors are permissible states in direct denotational semantics and in the Prolog formalism. The difference is that a denotational description must provide an explicit representation of possible error situations - an error is implicitly assumed in Prolog if any primitive fails to achieve its purpose. Each primitive will be discussed individually.

```
E[E']s == do(E')
C[C']s == do(C')
```

The action of these primitives is subsumed by the "do" predicate. It appears simpler, since it does not explicitly refer to the state with respect to which the denotation of the object is determined. The type of a language statement cannot be determined from the "do" predicate and reference to the actual action of the statement must be made.

```
TEST -> ACTION1 ,ACTION2  == or([TEST,ACTION1],
                                [ACTION2])
```

The "or" predicate is rather more general than the if-then-else construct used in this description, since it permits an arbitrary number of tests and actions. Since they can be mixed together, this weakens the concept of "test" in the prolog representation and any action which does not succeed may be regarded as a test.

```
NUM1 + NUM2  == plus(NUM1,NUM2,RESULT)
```

The only notable difference between these primitives is that the prolog form assigns a "name" to the result of the calculation, allowing it to be explicitly mentioned later in the description.

```
 =  ==  equall
```

As a test for equivalence of two items, these two primitives serve the same purpose. In the denotational description, however, this primitive is used to perform other tasks. In expression C4 above, for example, it is being used to assign values to v and s'.

```
 not
```

There is no Prolog equivalent of the "not" primitive. The effect of complementing a truth value is achieved by explicit manipulation of the booleans.

```
 isBool v == type(V,boolean)
 isNum  v == type(V,number)
```

For each domain, such as numbers, a function which tests an item for membership of that domain is generated. These are directly equivalent to the prolog "type" predicate which describes all the objects in the system.

```
 mI  == see(I,VAL)
```

Non-destructive examination of the value associated with an identifier is achieved by "seeing" the value in the environment. Notice that the prolog form has the side-effect of assigning a name to that item.

```
 m[v/I] == change([I,['$value$',v]]
```

The second argument to change may be an arbitrarily complex expression.

```
 null
 hd
 tl
```

The remaining primitives have no direct equivalent in the prolog form. The only larger data structure used in the denotational description is the list, which consists of a head and a tail which may be joined by a dot operator, or separated using "hd" and "tl". As was mentioned earlier, this notation for lists is essentially based upon a particular method for representing such structures in a computer.

The prolog representation represents all larger data structures as ordered sets. It uses a different set of primitives to access the structure. The choice of primitives is based upon certain assumptions about the way in which humans deal with ordered sets.

4.-Annotating the semantics.

At this point, we have described a semantic representation which provides an adequate method of describing the behaviour of programs. A new formalism has been introduced, but the system could be expressed in terms of Predicate Calculus.

In subsequent sections the tasks of modelling a learner and controlling an interaction will be discussed, as well as the problems identified in chapter 9. Each of these tasks makes use of the semantic notation which has been described. In most cases it is necessary to introduce extra domain-specific knowledge in order to solve the problems. It should be possible to formalize this additional knowledge.

The approach which will be taken is to extend the semantic representation such that specialized knowledge may be associated with each semantic primitive.

Reasoning processes may be guided by this additional source of information.

This knowledge will be represented using Annotated Predicate Calculus (APC) [Michalski 1983] which is a formalism that was developed for research on inductive reasoning. Michalski describes the overall aim of the formalism as follows;

> "..Annotated Predicate Calculus adds to predicate calculus additional forms and new concepts that increase its expressive power and facilitate inductive inference."

The main feature of APC is an ANNOTATION associated with each predicate,variable or function. These annotations contain problem-specific knowledge such as rules which relate that primitive to others, domain and range of functions, type of permissible descriptors etc. The annotations used in this tutoring system will be described as they are required in subsequent sections. To illustrate the idea, a partial annotation will be given here. The reader who wishes for more detail is referred to appendix 1 which lists all the information associated with each primitive.

```
Primitive: addbefore
Input:(1 1 1 0)
Type: (number o-set o-set o-set)
Rewrite: ((addbefore N A B C)
            ((addafter N-1 A B C))
```

This specifies that the first three arguments to ADDBEFORE must be given, while the fourth is a returned value. The first argument must be a number, while the rest are ordered sets. The rewrite rule specifies that attempting to add something before the Nth element of a set is the same as adding it after the (N-1)th element of the set.

5.-Applying the formalism to problems.

The previous section discussed the basic way in which knowledge about the semantics of a programming language are represented. In this section we will discuss the way that this formalism can be used to tackle five of the problems described in chapter 9.

The problems to be examined consist of; simulating an interpreter, generating descriptions of events, problem solving, problem generation and analysing program structure.

5.1-Simulating an interpreter.

Since there is no language interpreter as such in the system, a major task must be simulating the behaviour of an interpreter. This involves finding a way to mechanize the application of the semantic description to determining the effect of a particular expression. Within this problem we should examine the sub-problems of generating correct input-output behaviour, providing error messages at varying levels of complexity and generating information about intermediate states within a program. These issues will be discussed separately.

5.1.1-Mechanizing execution of expressions.

The problem involves using "effect" descriptions of language statements to imitate the behaviour of an expression composed of those statements. For example, the following behaviour

```
(QUOTE HERMIONE)
```

HERMIONE

should be produced by referring to the definition of QUOTE given in figure 19.

```
effect(quote(ARG1),
       [type(ARG1,'$sexpr$')],
       [out('$value$',ARG1)]).
```

Figure 19 - Semantics of QUOTE.

The problem is simplified by the fact that the syntax editor (as mentioned above) produces a parse-tree of the expression on which the rest of the program can operate. In the above instance this would be "quote(hermione)".

An automated interpreter must find the general schema which matches the expression it is trying to describe and instantiate that schema to produce a specific description of the effect of the given expression. In the above case, instantiation would produce

```
effect(quote(hermione),
     [type(hermione,'$sexpr$')],
     [out('$value$',hermione)]).
```

It was mentioned earlier that instantiation involves examining all the "effect" descriptions until one is found in which the abstract syntax and right-hand-side (if it exists) match the current expression.

It is not usually the case that a description is completely determined by the instantiation process. Normally there are still undetermined elements. To complete the description, each primitive must be examined in turn to ensure that its conditions of application are satisfied. In the above case this involves checking that "hermione" is of the specified type, and ensuring that there is no reason why output could not be generated. In this illustration all these tests succeed with no noticable effect on the description. A program to carry out this process has been implemented and appears with some sample interactions (including this one) in appendix 3.

It is worth providing one further illustration of this process for a more complex expression. The behaviour which we expect is;

```
(PLUS2 3 (PLUS2 4 5))

   12
```

where the "effect" which matches PLUS2 is

```
effect(plus2(ARG1,ARG2),
       [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARG1,'$number$'),
        do(ARG2),
        in('$value$',VARG2),
        type(VARG2,'$number$'),
        plus(VARG1,VARG2,RES),
        out('$value$',RES)]).
```

and the "effect" of a number is;

```
effect(NUM,
      [],
      [out('$value$',NUM)]).
```

The instantiation replaces all occurences of the variables ARG1 and ARG2;

```
effect(plus2(3,plus2(4,5)),
       [type(3,'$sexpr$'),type(plus2(4,5),'$sexpr$')],
       [do(3),
        in('$value$',VARG1),
```

```
        type(VARG1,'$number$'),
        do(plus2(4,5)),
        in('$value$',VARG2),
        type(VARG2,'$number$'),
        plus(VARG1,VARG2,RES),
        out('$value$',RES)]).
```

As in the previous case, each "type" test succeeds, so these can be removed from the description. When we reach the "do" primitive, it is essentially a request to determine the effect of a subexpression. For the purposes of illustration we will do this by substituting the meaning of that expression for the "do" primitive(the actual interpreter simply inserts a reference so that it is easier to generate debugging information). Instantiating the description of a number for the special case "3" ("type" predicates have been omitted for clarity), we get;

```
effect(plus2(3,plus2(4,5)),
       [out('$value$',3),
        in('$value$',VARG1),
        type(VARG1,'$number$'),
        do(plus2(4,5)),
        in('$value$',VARG2),
        type(VARG2,'$number$'),
        plus(VARG1,VARG2,RES),
        out('$value$',RES)]).
```

Examining the definitions of "out" and "in" shows that an out/in pair with the same channel name effectively cancel out, resulting in a substitution of the second argument to "out" wherever the second argument to "in" occured. This produces;

```
effect(plus2(3,plus2(4,5)),
       [type(3,'$number$'),
```

```
        do(plus2(4,5)),
        in('$value$',VARG2),
        type(VARG2,'$number$'),
        plus(3,VARG2,RES),
        out('$value$',RES)]).
```

The "type" test succeeds, and the "do" statement will be treated as in the previous case, leaving the description as

```
 effect(plus2(3,plus2(4,5)),
       [plus(3,9,RES),
        out('$value$',RES)]).
```

Finally, the plus primitive is found to be correct when RES is 12, and the corresponding substitution is made. Our description of the event is

```
 effect(plus2(3,plus2(4,5)),
        [out('$value$',12)]).
```

It can clearly be seen that this corresponds to the expected behaviour. In general this process reduces any description of an expression to instances of those primitives which have consequences for the state of the machine. In our system this is instances of OUT,CHANGE and CREATE. It is also worth noting that this mechanization will result in identical descriptions for events which have identical effects. If two expressions have the same effect their simplified descriptions will be the same, although the process of deriving those descriptions may be vastly different.

5.1.2-Generating error information.

As was discussed in chapter 9, there are several issues involved in the production of error information. One of these is the generation of simple "error messages" for the pupil, a second is the generation of more complex error information to help the teacher.

In the mechanization described above, an error occurs if one of the description primitives cannot be satisfied. For example,

```
(CAR (QUOTE ADRIAN))
```

would result in an error when the system tests to ensure that ADRIAN is a list. This violation of the requirements of a semantic primitive constitutes the basic component of all error information. This example and the semantic description of CAR will be referred to later. The description is;

```
effect(car(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
       in('$value$',VARG1),
       type(VARG1,'$list$'),
       element(1,VARG1,VAL),
       out('$value$',VAL)]).
```

This section indicates which semantic information could be used to produce error messages. It does not explain how this information is transformed to reasonable English. This is partially covered in subsequent sections, but constitutes an area of the system design which has not yet been examined in sufficient detail.

5.1.3-Simple error messages.

The task of producing simple messages which appear like those of a normal Lisp interpreter is quite simple. It will be recalled that these messages are intended to aid the pupil in transferring to a normal interpreter, so they should be concise and perhaps somewhat less informative than is optimal in a teaching environment (though they will be backed up by the teacher). A reasonable means of achieving this is simply to report which semantic primitive failed. Allowing for a canned message associated with each primitive this would produce something like;

"***** NOT A LIST"

for the above case. Few LISP interpreters are this unhelpful (there are some!). It is more useful to provide a little contextual information by giving details of the specific instantiation of the primitive. This would produce

"***** ADRIAN IS NOT A LIST"

This is the level of message chosen for the interpreter. More details of the actual messages used will be found in appendix 1.

It is clear that neither of these messages is adequate for anything except reminding an expert user of a careless slip. This is why it is necessary to produce more detailed information for the teacher to use in guiding the pupil.

5.1.4-Detailed error information.

The first advantage that the teacher has is detailed knowledge of the location of an error. It is apparent which statement description was being used when the error occured, and where the primitive which failed was located in that description. In the above example this is sufficient to indicate that CAR expects the value of its argument to be of "type" list, and that this is what ADRIAN failed to satisfy.

As well as examining those things which occured before the error, useful information can often be produced by continuing execution from the error point and detecting any consequences of the error. In the case shown above this would result in a second error since it is not possible to take the first element of ADRIAN. This provides sufficient information for a more detailed description of what went wrong, such as;

> " CAR evaluates its argument. It expects the result to be a LIST. It examines the first element from this LIST. The value of the argument is ADRIAN. ADRIAN is not a LIST. It is not possible to examine the first element of ADRIAN."

It may be the case that more than one error is generated by a particular expression. In this case it is important to identify all the errors which occur so that the teacher can decide which ones to talk about and what order to discuss them in. Sometimes the error messages refer to the least important problem with the expression!

Continued execution will produce a set of violated semantic primitives, without distinguishing those which are basic errors from those which are contingent upon other errors. In the above example, the violation of "type" and of "element" should be grouped together since correcting one will also correct the other.

While it is possible to approach this problem by complex reasoning about the expression, the method of separating errors adopted here is somewhat simpler. It is often the case that a programmer tests her work by inserting typical values at various points in the execution; a similar technique will be used here.

When an error occurs, the semantic interpreter may react to it by simply continuing execution or by aborting. A third possibility is to introduce a result which would be typical for that primitive if it had not failed. Continuing execution will now produce only errors which are not contingent upon the first. Some examples of this appear in the appendix. The following example shows the behaviour of (CAR(QUOTE ADRIAN)), both without and with patching;

WITHOUT PATCHING;

```
| ?- do(car(quote(adrian)),M).

1) cannot satisfy;
type(adrian,list)
2) cannot satisfy;
element(1,adrian,_39)

M = [[$value$,_39]]

yes
```

WITH PATCHING;

```
| ?- do(car(quote(adrian)),M).

1) cannot satisfy;
type(adrian,list)
  ### Patched (a,b,c) for adrian

M = [[$value$,a]]

yes
```

It is important to realise that this technique does not always produce useful information. The choice of patched item is very important. In some cases the patch may be sufficiently inappropriate to generate more errors than it corrects. For example;

(MAPCAR X 'ADD1)

requires that X have a list of numbers as its value. The default patch if the value is not a list would be to substitute a list of atoms. This will cause ADD1 to fail because it does not receive a numeric argument.

Another task for the interpreter to satisfy is the execution of partially complete expressions in such a manner that errors due to incompleteness can be distinguised from "genuine" errors. This problem has not been explored in depth, but it seems likely that the patching technique described above can be applied to this task.

At present, the interpreter shown in Appendix 3 only carries out patching at a rudimentary level.

5.2-Generating information about execution.

A teacher does not operate only when the student makes an error. There must be a means to collect information during the execution of an expression. This information can be used to decide what to tutor and to refer to concrete instances of various issues.

The method applied in this system is somewhat similar to the use of a trace package in a normal language intepreter. In an interpreter, tracing information is normally generated at entries and exits of language statements. In this system the tracing is at the level of semantic primitives.

As an example, consider a discussion of evaluation. Within this discussion it would be useful to provide an illustration. Suppose that the pupil has recently executed the expression

```
(CONS (QUOTE A) (CAR (QUOTE ((B C) D))))
        (A B C)
```

The system has information associated with its representation of evaluation which indicates that "do" and "out($value$,M)" are relevant to discussions of evaluation. Repeating the execution of the above expression while "tracing" these primitives would produce;

```
CONS:
  do(quote(a))

      QUOTE:
        out($value$,a)

  do(car(quote(**list(**list(b,c),d)))))
      CAR:
         do(quote(**list(**list(b,c),d)))
           QUOTE:
             out($value$,**list(**list(b,c),d))
         out($value$,**list(b,c))

  out($value$,**list(a,b,c))
```

Figure 20 - Tracing output.

The information shown here could be used to generate output for the pupil such as;

> " CONS evaluates its first argument, (QUOTE A), to get A. It evaluates its second argument, (CAR(QUOTE((B C) D))) to get (B C)."

The language generation problem will be discussed in the next section. The sort of "tracing" information associated with each topic will be discussed in the section on interaction.

5.3-Describing statements and programs.

As has been shown in the previous section, the semantic interpreter can be used to generate information about expressions in the language. These can be reasoned about by the system, but ultimately there comes a point at which information of this type must be presented to the pupil. This involves providing some language generation mechanism based upon the formal semantic representation.

There has been some detailed research on the problems of language generation (e.g. [Davey 1978]). It is beyond the scope of the current project to attempt to make a new contribution to this area. Instead, the system is provided with a simple method of producing highly stylised English. The description of this system is included for the sake of completeness.

Two related problems will be discussed - generating help information and describing program behaviour.

5.3.1-Help information.

A simple package for generating help information has been implemented. Appendix 4 shows some sample output. It operates by taking a canned phrase associated with a particular primitive, and filling in any undetermined components with the corresponding arguments of the particular instance which it is attempting to describe. For instance, by combining the description of QUOTE in figure 19 with the following canned phrases,

```
(type A B) => " It checks that A is of type B"
(out A B) => " It returns a A which is B"
```

we can produce a description of the general behaviour of QUOTE which reads;

```
It checks that ARG1 is of type $SEXPR$.
It returns a $VALUE$ which is ARG1.
```

If we had a specific instance of QUOTE, such as (QUOTE HANGOVER), exactly the same mechanism would produce

```
It checks that HANGOVER is of type $SEXPR$.
It returns a $VALUE$ which is HANGOVER.
```

This technique produces reasonably acceptable descriptions, although they are sometimes rather obtuse, as the following summary of COND illustrates;

```
It checks that ARG1 is of type $SEXPR$.
It checks that ARG2 is of type $SEXPR$.
It checks that ARG3 is of type $SEXPR$.
It checks that ARG4 is of type $SEXPR$.
It evaluates ARG1.
It retrieves the $VALUE$ and calls it VARG1.
At this point one of two things happens:
 EITHER A test is made to see if VARG1  is the same as T
        It evaluates ARG2.
        It retrieves the $VALUE$ and calls it VARG2.
        It returns a $VALUE$ which is VARG2.
 OR,  It evaluates ARG3.
      It retrieves the $VALUE$ and calls it VARG3.
      At this point one of two things happens:
       EITHER A test is made to see if VARG3 is the same as T
               It evaluates ARG4.
               It retrieves the $VALUE$ and calls it VARG4.
               It returns a $VALUE$ which is VARG4.
       OR,  It returns a $VALUE$ which is NIL.
```

This is clearly an inadequate mechanism for generating teaching statements, since these must be controlled by the current state of the pupil and the overall teaching strategies as well as simple domain knowledge. Some indication of the way that these links are achieved is given in chapter 12.

5.3.2-Describing program behaviour.

If the mechanisms described in the previous section can be relied upon to select the relevant information from a particular execution, the problem of describing aspects of a program is only slightly more difficult than providing help information. Applying the canned phrase technique to the set of primitives shown in figure 20 would produce

> " It evaluates (QUOTE A). It returns a value which is A. It evaluates (CAR(QUOTE((B C)D)). It evaluates (QUOTE((B C) D)). It returns a value which is ((B C) D). It returns a value which is (B C). It returns a value which is (A B C)."

It can be seen that this is not an adequate description since it misses something vital about the relationships which hold between the components. There is a need to incorporate some contextual information, giving a description such as;

> " In order to evaluate (CAR(CDR X)) we need a value for (CDR X). In order to evaluate (CDR X), we need a value for X. X is evaluated. It returns the value (A B C). (CDR X) returns the value (B C). (CAR (CDR X)) returns the value B."

This higher level of language generation has not been provided in IMPART.

5.4-Generating and Solving programming problems.

The area of automatic programming is very large. It is not possible to survey the relevance of all this work within the current system, and such an activity would not be central to the design in any case.

Instead, this section describes the task which a problem-solver must tackle within a tutoring system. The links between problem-solving and problem-generation are discussed. Finally, a simple problem-solving program which uses the formal semantic representation is described.

5.4.1-Special requirements.

As was mentioned earlier, the problem-solver must generate solutions of a form which the pupil can understand. This requires the problem-solver to make use of the pupil model in order to decide which currently focussed information may be applied to the problem. The problem-solver must also use techniques of solution which the pupil understands if the output is to be directly used as a tutoring aid.

In this section an illustration of the behaviour of a problem-solver which uses the semantic representation will be given. This example applies the problem-solving techniques to a correct description of LISP. For a given pupil, the problem-solver may be applied to the semantic definition which that pupil currently believes corresponds to LISP (see chapter 11 for the derivation of this definition). This provides a method for adapting problem solutions to the current state of the pupils knowledge.

5.4.2-A simple problem-solver.

A program to generate problem-solutions is listed in appendix 5. This makes use of the formal semantics together with some annotations in order to generate a problem-solution. To show its operation, consider the following simple problem;

> " Write an expression which extracts the third element of the list (A B C)."

The first task is to solve the problem at the semantic level. A formal version of the problem specification would be

```
element(3,**list(a,b,c),RESULT)
```

To determine which semantic primitives are available in this language, the language definition is examined for functions containing items which appear in the problem specification. In the above case, with the definition of LISP in appendix 2, we find that CAR is the only relevant function, and that it only allows "element(1...)" to be selected. Now one of the annotations of "element" contains a rewrite rule which specifies that

```
element(N,X,M)==after(J,X,K),element(N-J,K,M)
```

so this suggests that language statements containing "after" may also be relevant. Scanning the language definition for these statements produces CDR which allows "after(1...)".

At this point the problem has become finding a combination of "element(1...)" and "after(1...)" which corresponds to "element(3...)". The rewrite rule allows us to produce the expression;

```
element(3,**list(a,b,c),M)
    ==after(1,**list(a,b,c),**list(b,c)),
      element(2,**list(b,c),M)
```

Reapplying the rewrite rule to the second element of this description produces the semantic level solution;

```
element(3,**list(a,b,c),c)
    ==after(1,**list(a,b,c),**list(b,c)),
      after(1,**list(b,c),**list(c)),
```

```
element(1,**list(c),c)
```

The problem which remains is to express this in terms of the language statements from which the semantic primitives were taken. This involves establishing communication channels between the constants in the description - either as explicit channels or via variables.

The definition of CDR is

```
effect(cdr(ARG1),
      [type(ARG1,'$sexpr$')],
      [do(ARG1),
      in('$value$',VARG1),
      type(VARG1,'$list$'),
      after(1,VARG1,VAL),
      out('$value$',VAL)]).
```

Since an "after" acts directly on the original data we may begin by instantiating the "after" primitive. This leads us to the conclusion that the data must be received by CDR on the $value$ channel. This is equivalent to saying that CDR must have an argument which evaluates to **list(a,b,c). This leads to searching for a solution to this new problem, which results in the expression (QUOTE (A B C)). This will be inserted as the argument to CDR. Similar methods can be applied to instantiating the other function descriptions, and eventually a solution will be generated;

```
(CAR (CDR (CDR (QUOTE (A B C)))))
```

If the LISP definition had included a semantic description of the LAST function, the program would (also) have generated

```
(LAST (QUOTE (A B C)))
```

It should be borne in mind that this program is simply a demonstration of an approach to problem-solving. It has not been shown to be capable of solving many problems. It should be the case, however, that since this approach searches for solutions using the semantic primitives, changing the semantic description will allow the problem-solver to produce solutions in other programming languages.

5.4.3-Problem generation.

The teacher should be able to generate problems for the pupil to solve which give practice in particular areas or bring certain features of the language to the pupils attention. This involves producing a problem specification which can then be filled out to give a complete problem which can be presented to the pupil and a solution for that problem.

The generated problems must have solutions which are within the current capabilities of the pupil. For this reason it seems sensible to use the problem-solver as part of the problem generation process. A proposed mechanism is to allow the problem solver to insert "typical" problem components in a partial problem specification until a problem which it can solve has been produced.

As an example, suppose we wish our pupil to practice addition. Our problem specification would be that we wish to use instances of the PLUS primitive. The problem-solver may scan the language definition for statements which include this (ADD1 and PLUS2 in the case of LISP). Since our problem placed no constraint on the primitive arguments, we may generate problems such as;

" Write an expression which adds 3 and 4"

or

" Write two different expressions which add 1 to 23" and the solution which has been generated by the system can be compared with the attempts of the pupil to solve the problem.

5.5-Patterns within a program.

In this section a brief illustration will be given of the way in which program slicing can be implemented as a debugging aid. An attempt will also be made to show how these techniques can be applied to linking program structures with the semantics of individual statements.

5.5.1-Slicing.

Weiser defines a slice in terms of a particular variable in a particular statement of a program. The slice corresponds to an executable program from which all those expressions which do not affect the current state of the variable have been deleted. This deletion is achieved by performing a data-flow analysis on the program. Zislis uses a similarly abstract approach based upon manipulations of a directed-graph form of the program.

Within our formalism a "slice" can be produced without eliminating parts of the program, by simply focussing on certain aspects of the system. For example, where Weiser requires a large amount of computation to slice with respect to a variable X at a particular point, we can, at the simplest level, simply look for occurences of CREATE,DELETE,CHANGE and SEE within our program. Consider the following example;

```
(PROG (X L)
     (SETQ X Y)
     (SETQ L NIL)
LOOP (COND((ZEROP X)
                (RETURN L)))
```

```
          (SETQ X (SUB1 X))
          (SETQ L (CONS 'Q L))
          (GO LOOP))
```

a slice with respect to variable L would give

```
     (PROG (X L)
          (SETQ L NIL)
          (COND((ZEROP X)
                    (RETURN L)))
          (SETQ L (CONS 'Q L)))
```

This slice is executable, but its relation to the overall program is not clear. If, on the other hand, our program monitors the relevant primitives during execution of the complete program then a more meaningful behaviour results. This is more directly explicable to the pupil.

The slicing mechanism used by IMPART is not constrained to slicing upon variables: for example, dataflow slices could be made by monitoring IN and OUT statements, while control flow slicing could be achieved by monitoring DO statements.

The extent to which the slicing mechanism is operational within IMPART is limited by the fact that higher level structures than program statements are not currently well represented (see below). The implementation does not make use of slicing techniques. I wish to suggest that slicing is an important mechanism which can relate low-level semantic knowledge to knowledge at the level of algorithms. This should form a focus for further research.

5.5.2-Higher levels of representation.

In any programming language there exist commonly used groupings of statements intended to achieve particular goals. These higher level structures may be regarded as general purpose algorithms which may be instantiated in particular ways to solve particular problems. Being able to recognise and apply such constructs is part of learning to use a language.

These structures are often too complex and difficult to be discovered by experiment alone. If we wish to teach effectively then we must be able to guide our pupil towards finding these structures. We must consider how to represent them and how to guide the pupil towards them.

It is possible to provide the system with a built in set of plans which it may tutor, such as those used by the Programmers Apprentice [Rich 1981] . Alternatively, we could attempt to provide the system with mechanisms which allow it to recognise good plans when it sees them, and to create abstract plans by analysing particular solutions to problems. The latter task seems extremely difficult if the system has no guidance as to how plans are generated.

It has already been pointed out that the teacher should avoid simply stating a rule without being able to explain it. One advantage of the second technique is that the derivation of the plan incorporates information which can form the basis of an explanation to the pupil.

In general, then, we would like to provide the system with rules which it can use to derive language structures. These rules should build upon a knowledge of the language semantics. The derivation should form the basis of a "psychologically reasonable" explanation of the purpose of the structure. The system may be provided with plans of actual structures in order to guide its plan search, but it should be able to produce a derivation for these plans even if it knows the result.

An example would be that iteration should be derivable from the following piece of code by appealing to some goal of achieving parsimony of code;

```
write( 'hello');write( 'hello');write( 'hello')
```

becomes

```
for i:=1 to 3 do write('hello')
```

5.5.2.1-High level structures in this system.

The goal of deriving plans from lower level information about the language is a difficult one to fulfill. It has not been possible to tackle it within the scope of this project. Since the process of deriving plans is considered to be important there has been no attempt to produce a large library of unexplained plans such as that which exists in the Programmers Apprentice. Instead, one plan has been produced to illustrate the sort of information which the system is expected to handle. This is an area which requires further investigation.

5.5.2.2-Simple recursion.

In this section we will consider a simple three clause recursive Lisp function and show how it relates to a general plan for such functions.

```
(DE DOWHAT (A L)
    (COND ((NULL L) **1**
             NIL)   **2**
           ((EQUAL (CAR L) **3**
                  A)
             (DOWHAT (CDR L)))
           (T              **4**
             (CONS (CAR L)   **5**
```

```
(DOWHAT (CDR L)))) ))   **6**
```

1 is known as the stopping condition. This is a case for which a definite result can be given without having to do any more recursion. The clause with the stopping condition always appears first. The type of stopping condition is related to the type of arguments and the form of the simplification function (see below). For example, taking a list apart with CDR requires a NULL function for a stopping condition, while counting down a sequence of numbers would require ZEROP.

2 is the stopping value. This is the value which is returned from the deepest invocation of the function when the recursion terminates. The type of value returned here must be suitable for the building function (see below) since it will become an argument to that function. For example, if the result is being built with CONS then the stopping value must be a list. If the result is being built with PLUS then the stopping value must be a number.

3 is a clause dealing with a special case. It carries out some action if its test succeeds. Not all functions have a clause like this, but some may have lots of them.

4 is the clause which operates when none of the special cases succeed. It is generally used to break up the problem and rebuild the solution.

5 indicates the CONS function which is the function used to build the solution in this particular case.

6 is the recursive call (i.e. the call to the function currently being defined). Note that the recursive call must include a simplified version of the original argument to the function. In this case the simplifying function is CDR. The simplifying function must be such that repeatedly applying it to an example will eventually produce an expression that will satisfy the stopping condition.

The following corresponds to a first attempt at representing the relationships between these components in a plan.

ROLES: stopping-condition,stopping-action,
simplifying-action,result-building-action,
recursive-step

INPUTS: argument1,argument2...

OUTPUTS: value

RELATIONS:

1) Repeated application of simplifying-action to argumentN results in stopping condition becoming true.

2) Stopping-action produces a value which is a legitimate argument to result-building-action.

3) The value of simplifying-action applied to argumentN must occur as argumentN of the recursive step.

6.-Summary of chapter 10.

A method of representing the semantics of a programming language has been described. This representation maps directly into PROLOG. Mechanisms to tackle the problems outlined in chapter 9 have been discussed. Some issues, involving high level representations of programming structures and "conceptual slicing" of programs, have been raised without being solved. This chapter has outlined the core of the domain-specific reasoning facilities in IMPART.

Chapter 11.

User-modelling

and

Machine Learning.

The central part of a model of the pupil is a model of the learning process. In this chapter, the mechanisms used by the teacher to represent the learning process will be discussed. Following this, brief outlines will be given of other points in the system at which information about the user is stored.

1.-Modelling the learning process in a practical system.

1.1-A framework for learning.

In this system we will attempt to move "..from an expert-based to a learner-based paradigm" [Goldstein 1982] . An expert-based paradigm would involve giving a central position to an "overlay" model of the learner, and attempting to equate this model with some predetermined "expert" level. As has been shown, such an approach is inadequate for achieving a high level of individualised tuition. To operate within a learner-based paradigm, we must concentrate on the problem as seen by the user, and apply a theory of learning to the task of expanding the users ability to control the environment.

Our aim is to provide the pupil with a predictive model of the environment which is derived from observations which the pupil can make. This process is achieved by learning mechanisms, and it is these mechanisms which the teacher must model and guide.

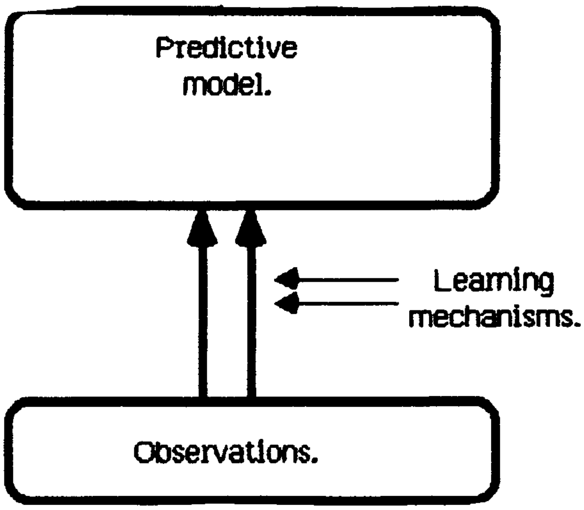


Figure 21 - A learner based pupil model.

More specifically, we wish our pupil to apply inductive reasoning to concrete examples in order to produce a model of the domain. It should be possible in turn to apply deductive reasoning to this model in order to generate concrete predictions about the domain.

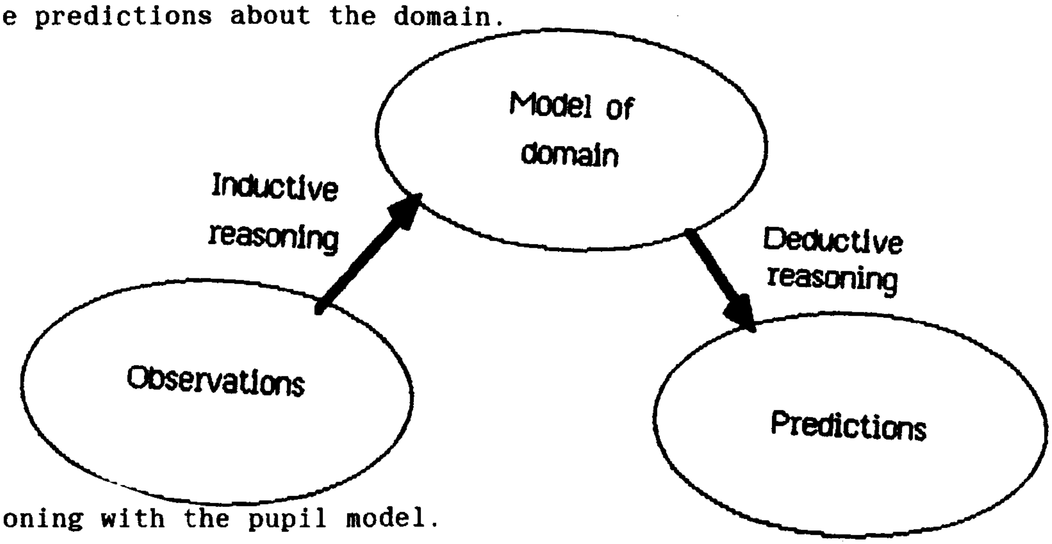


Figure 22 - Reasoning with the pupil model.

By taking this approach, our teacher is prevented from stating results which the pupil must memorize without understanding. The pupil is gaining knowledge rather than facts. Our teacher must aid this learning process in three basic ways;

1) PERCEPTION. The pupil must learn to perceive the environment in a structured way. Certain things are important and others are irrelevant. A novice will perceive "(CONS A B)" as a string of characters which is not significantly different from "(CONS (A B" . An expert will perceive the

expression in terms of functions and arguments with brackets as delimiters: this immediately makes the latter example senseless.

2) GENERAL LEARNING. There are certain general techniques of learning which may be applied in many environments. It may be necessary to instruct the pupil in how to apply these techniques in a given environment. For example, if the student knows how to turn constants into variables, she should learn that this method may reasonably be applied to all arguments to a LISP function, but will not yield sensible results if applied to a function name. An example of this would be that generalizing from

```
(QUOTE ALICE)
  ALICE
```

to predict

```
(QUOTE FRED)
  FRED
```

is reasonable, but to predict

```
(ADD1 ALICE)
  ALICE
```

is clearly not a sensible generalization. Heuristic guidance of this kind may be associated with both inductive and deductive reasoning.

3) DOMAIN SPECIFIC LEARNING. Some techniques of learning may be available which apply only in the restricted domain being tutored. The teacher should introduce the pupil to these techniques. In our case, where our goal is to make the pupil independent of the teacher, we wish the pupil to continue learning from a normal language interpreter/compiler. This requires us to tutor the strategies necessary to acquire useful information from simple error messages.

In the following sections we will make use of three major objects in our learning framework; Facts, Concepts and Predictions. If we refer back to figure 22, these items may be associated with different areas on the diagram.

FACTS are observations about the world. They need not be derived from anything else, and do not embody any predictions about the way that the world operates. They could be held as a table of events to look up. This is the lowest level of understanding which can exist. "(QUOTE FRED) returns the value fred." is a fact - it corresponds to a description of an event in the world (where the world is a programming environment). We cannot answer the question WHY in terms of any more primitive units (for example " An opening bracket appeared on the screen, a letter Q appeared on the screen ..."). The expression returns a value because that is the way things are in Lisp. If asked why the value is FRED, we would actually appeal to a more complex piece of information, by offering the general rule that QUOTE returns its first argument as its value. In practical terms, not all facts can be derived from a non-intelligent environment. In the above case, for example, the pupil would discover what is returned by an expression, but would not know that such a statement is conventionally called a value. If this secondary sort of fact is considered necessary (which it would be if our pupil is to talk to others about Lisp) then such facts must be imparted by the teacher. In figure 22 facts are descriptions of the concrete events which occur in the environment.

CONCEPTS are abstractions which represent sets of facts in a more concise form. They may be regarded as organisers of the domain. "Lisp functions return values" is a concept because it explains part of the behaviour of every function, whereas storing these as separate observations is less efficient. Concepts provide a generative representation for pieces of domain knowledge. The pupil model consists of a set of concepts which the pupil may be using to describe the domain.

PREDICTIONS are like facts in that they represent concrete events in the world. The difference is that they are derived from the pupil's conceptual model of the world, and as such must be validated by comparison with the FACTS. When a prediction is validated it gives more weight to the abstractions on which that prediction was based.

In order to tutor according to this model of learning it is necessary to find a formal representation capable of expressing the derivation of "concepts" from "facts" by various routes.

In this case, our "facts" consist primarily of observations which the pupil can make about the behaviour of the language interpreter. These facts are augmented by statements made by the teacher. If we identify "facts" with directly observable features of language expressions and their consequences, we may identify "concepts" with abstracted representations of the behaviour of particular language statements or abstractions whose predictive power spans several such statements.

The representation used for facts and concepts is essentially an augmented version of that used to describe the semantics of the language being tutored. Each interaction step is represented by describing observable features in this formalism. Individual statements are described by building a formal semantic description of the language the user thinks she is using. More general concepts are constructed from combinations of the semantic primitives.

The relationships between facts and concepts are determined by a model of the learning process which seeks to link them via a combination of general learning rules guided by domain-specific heuristic knowledge.

In the following sections we will discuss inductive and deductive mechanisms for reasoning. These mechanisms are used to link facts to concepts and concepts to predictions, respectively.

As was discussed in chapter 4, there is no existing theory of learning which would adequately describe the behaviour of a human LISP learner (although this area is currently being explored [Anderson 1982]). Rather than attempting to produce such a theory, the approach which has been taken here is to adopt some recent Artificial Intelligence research on learning and to focus on the role of a learning model rather than on its detailed structure. The model of concept acquisition which is applied is fairly simple, and cannot be expected to correspond to a model of what goes on in human learners. It is simply claimed that, in this restrictive domain, it is capable of making inferences which are similar in effect to those of a human, and provides some bounds to the set of hypotheses which the pupil may hold. The differences between real pupils and this learning mechanism may provide an interesting area of study.

1.1.1-Inductive reasoning.

Inductive reasoning provides a means of going beyond the predictions which a rigorous logical system would make from a given set of data. In inductive generalisation a hypothesis may be produced which is falsity-preserving (i.e. if the data was false then the hypothesis is false), but does not necessarily preserve the truth of the original data. To quote Dietterich [Dietterich 1983] "A generalization rule does not guarantee that the obtained description is useful or plausible."

The task which we must approach is to generate hypotheses about the behaviour of Lisp expressions from observations, and to provide some assessment of the usefulness and validity of these hypotheses. The formalism chosen to tackle this problem is drawn from Artificial Intelligence work on "Machine learning".

More specifically, the system described here may be seen to have much in common with Michalski's Annotated Predicate Calculus [Michalski 1983] .

It may seem unreasonable to apply a model developed for achieving learning in machines to the modelling of human learning. This is not the case since the ultimate yardstick by which such models are assessed is their similarity to human learning. As Michalski observes;

> " The results of computer induction should be symbolic descriptions of given entities, semantically and structurally similar to those a human expert might produce observing the same entities. Components of these descriptions should be comprehensible as single "chunks" of information, directly interpretable in natural language, and should relate quantitative and qualitative concepts in an integrated fashion."

A further objection that might be raised is that Machine theories of learning are not yet in a sufficiently advanced state to support this type of application, and have not been shown to have any psychological validity. The bounded user-modelling technique described here requires only that the theory is sufficiently powerful in the range of hypothese it can generate; it need not be identical with the actual method of learning used. It is not clear that human teachers always have appropriate models of the learning mechanisms of their pupils. As Simon [Simon 1983] points out,

> " It is a salient characteristic of human learning procedures that neither teacher nor learner has a detailed knowledge of the internal representation of data or process."

The learning task which we wish to model is rather simpler than that tackled by large learning programs such as AM [Lenat 1983] . We have several simplifying conditions;

1) The pupil will only proceed in small steps from her current state of knowledge. Vast and complex inductions are beyond the scope of our pupil so we may drastically restrict the search space of possible hypotheses. If the teacher could make such inductions then she could plan on a larger time-scale, whereas short look-ahead plans may result in both teacher and pupil exploring a blind alley. This consideration will actually lead to a trade-off between complexity and simplicity in the look-ahead strategies.

2) There will normally be a "focus" to the learning activity. A particular problem will be central over a period of time, and during that time it will be a major component of all generated hypotheses. An example would be learning to extract elements from lists. In this case the focus is clearly defined as the ordered-set access functions of the semantics.

3) There exists a frontier of student knowledge. Certain concepts will be stable and well understood while some will be in a more tenuous state. The latter define a "frontier" of student knowledge and the teacher will typically involve the student in work which is upon that frontier.

1.1.2-Selective induction.

Selective generalization involves representing a domain in terms of a given set of primitives. The selection comes in deciding which of these primitives are important in a particular situation as organisers of the input knowledge.

As an example, suppose we present a system with a red square, a red triangle and a red circle. The system might represent these as

1) shape(square),colour(red)

2) shape(triangle),colour(red)

3) shape(circle),colour(red)

This is simply a description of each item - a set of facts about the world. Our system might apply an inductive reasoning rule to these facts, such as dropping a condition if it does not occur more than once. In the above case all the shapes are different, so the rule would give the result "colour(red)". This corresponds to the suggestion that the only unifying thing about these examples is their colour.

1.1.3-Constructive induction.

Constructive generalization is the process of generating new descriptions which have primitive elements which are new - being derived from simpler primitive elements. For example, given three rectangles of sides 2*6, 4*3 and 1*12, respectively, we could describe them in terms of sidelength primitives as;

i) sidelength(2),sidelength(6)

ii) sidelength(4),sidelength(3)

iii) sidelength(1),sidelength(12)

but this description does not seem amenable to any unifying generalizations. If we could construct a new descriptor "A", such that any object of sidelength(X),sidelength(Y) could be redescribed as A(X*Y), then the above events would ALL be instances of A(12). In effect we have discovered the concept of area.

It will be observed that this process involves two steps. The first is designing a new descriptor and the second is applying it to the actual data. It seems likely that the two processes are related. In the above case our search for a new descriptor would be driven by a goal of explaining all the observations.

IMPART actually generates new descriptors by applying pattern matching to sets of complete descriptions which it has produced. In IMPART this mechanism has only been provided at a rudimentary level. The system has succeeded in generating one new descriptor. It found that many descriptions of Lisp functions of the form (fn arg1 ... argn) involve the sequence of statements "do(arg1)...do(argn)". A new descriptor was generated to represent this. This corresponds to "discovering" the concept of an eval-spread function in LISP.

1.2-An algorithm for learning.

We will now apply rules of inductive reasoning to the sort of observations a student could make when interacting with a language environment. The primitives on which the rules operate are the same as those used in the formal semantics of the language.

There are three basic steps to the learning strategy. The first of these is perceiving the available information, the second is generalizing to produce abstractions from the representation, and the third is transforming those items which are inadequate descriptions using domain specific knowledge.

1.2.1-PERCEPTION.

Perceiving the events which occur is not necessarily as simple as it appears. Part of the problem of understanding a new domain is learning how to classify observations. For a novice Lisp programmer (QUOTE AUNTIE) is not obviously a function call; she has no reason to perceive AUNTIE and QUOTE as different sorts of object. In this mechanisation the problem is overcome by providing some rules to aid the program, but such rules are not available to the pupil. The teacher must impart this knowledge to the student. As an example, consider the following instance of the QUOTE function;

```
(QUOTE (A B C))
 (A B C)
```

The system notes everything it can perceive. In our representation this corresponds to inserting instances of OUT,CHANGE,DELETE and CREATE;

```
quote(**list(a,b,c))
  [out($value$,**list(a,b,c))]
```

A heuristic indicates that typing information is always a relevant part of the perception of an expression (this must be taught), so every possible item is assigned a type (in general, any item which can be named is given a type);

```
quote(**list(a,b,c))
  [type(quote(**list(a,b,c)),sexpr),
   type(**list(a,b,c),list),
   out(value,**list(a,b,c))]
```

"The function QUOTE was applied to the argument (A B C). This is an sexpression. (A B C) is a list. A value (A B C) was returned."

This corresponds to a complete description of the observed event.

1.2.2-GENERALIZATION.

The above description only represents the behaviour of a particular observed event. Its predictive power is only sufficient to tell us what will happen if we execute an identical expression. If we wish to produce a more powerful representation of the domain than an enumeration of every possible event, we must attempt to derive a more general rule from the specific one.

In the above case we may apply the rule for turning constants into variables to every constant which occurs more than once. This gives the following generalization;

```
quote(ARG1)
  [type(quote(ARG1),sexpr),
   type(ARG1,list),
   out(value,ARG1)]
```

" The function QUOTE is applied to an argument. This is an sexpression. The argument is a list. The argument is returned as a value."

This is almost a correct description of the action of QUOTE. It is a slight specialization in that ARG1 may be any s-expression. If a further example which took an atom were given, the system would produce the following generalization;

```
(QUOTE A)
    A

quote(ARG1)
  [type(quote(ARG1),sexpr),
   type(ARG1,atom),
   out(value,ARG1)]
```

Comparing the generalizations of these two events, we find that they differ in the permissible type for the first argument. Since the system is provided with a hierarchy of language datatypes (which must be taught directly), it is possible to apply the "climbing generalization tree" rule to show that these may be generalized to type SEXPR.

1.2.3-TRANSFORMATION.

The relationship between constants of a description is not normally as clear as in the above example. If we are to produce the most general possible description, then we must be prepared to search for more devious relationships between components. This involves using domain knowledge to decide which primitives of the representation can transform their arguments, what sort of argument can be transformed, and when such a transformation is appropriate. For example, addition is a transformation which may be applied to numeric arguments if you are seeking to relate an unexplained number to some existing numbers. In another case, the same number may be related to a list by extracting an element from that list. This sort of information is kept in the annotation associated with each descriptor.

To illustrate this consider an example of SETQ, where we assume that our system already knows about evaluating atoms;

```
(SETQ Z B)
   5
```

Observation of this produces a more complex description than in the previous case, because our system can "see" the side-effect of assigning 5 as the value of Z, which results in the use of a "change" statement.

```
setq(Z,B)
[type(setq(Z,B),sexpr),
 type(Z,atom),
 type(B,atom),
 change(Z,5),
 out(value,5)]
```

We may apply the generalization rule for constants to variables, as in the previous example;

```
setq(ARG1,ARG2)
[type(setq(ARG1,ARG2),sexpr),
 type(ARG1,atom),
 type(ARG2,atom),
 change(ARG1,RES),
 out(value,RES)]
```

Because 5 was a repeated constant, it has been into a variable (the name RES is arbitrary). At this point we notice a problem because we have introduced a variable which is unrelated to anything. There are two possible courses of action. The first is to turn RES back into a constant, leaving SETQ as a function which always returns 5 (and assigns 5 to its first argument). The second is to attempt to transform the constants of the original problem in order to establish a link between RES and some other constant. We find that the "do" descriptor is a prime candidate for such a transformation. If we return to the specific instance and apply this transformation to the description, we produce a new description;

```
setq(Z,B)
[type(setq(Z,B),sexpr),
 type(Z,atom),
 type(B,atom),
 do(B),          ** change
 in(value,5),    ** change
 type(5,number), ** change
 change(Z,5),
 out(value,5)]
```

We may now repeat the generalization stage, which will produce a reasonable guess at the behaviour of SETQ. It is not actually correct, because it does not include information about variable declarations, but it provides a basis for tutoring such difficulties.

1.2.4-Induction across examples.

Let us assume that our system has now mastered some basic expressions and we provide the following example of COND;

```
(COND (X Y)
      (T Z))
```

If, in our first example X is true, then we would expect the system to hypothesise

```
cond(ARG1,ARG2,ARG3,ARG4)
    [do(ARG2),out(ARG2)]
```

Giving the system an example in which X fails would produce the description,

```
cond(ARG1,ARG2,ARG3,ARG4)
    [do(ARG4),out(ARG4)]
```

Our system clearly has a problem in combining these. The simplest strategy is to link them using a disjunction;

```
cond(ARG1,ARG2,ARG3,ARG4)
    or([do(ARG2),out(ARG2)],
       [do(ARG4),out(ARG4)])
```

This may become a working hypothesis about the behaviour of COND. It is apparent that the behaviour of this statement would require the system to observe many events before a correct hypothesis could be generated. This illustrates the way in which errorful student hypotheses can be included in the pupil model.

1.2.5-A complete set of rules of generalization.

The list of rules which are actually used by the learning system is given in this section. They are a subset of a taxonomy provided by Michalski [Michalski 1983] .

1) DROPPING CONDITION - Given a description of an event, a more general representation can be reached by simply eliminating one component of the description. In the domain in which we are interested the application of this rule may be constrained by heuristics which recognise circumstances in which the rule can be appropriately used.

2) ADDING ALTERNATIVE - If two alternative events belong to the same category, then they may be combined by using the disjunction operator to link them. For example, given these events;

i) if_then_else(ARG1,ARG2,ARG3)
    [do(ARG1),in($value$,VARG1),equall(VARG1,t),do(ARG2)]

ii) if_then_else(ARG1,ARG2,ARG3)
    [do(ARG1),in($value$,VARG1),equall(VARG1,nil),do(ARG3)]

we may combine them by noting the differences in the description and inserting a disjunction, resulting in the description

```
[do(ARG1),in($value$,VARG1),or([equall(VARG1,t),do(ARG2)],
                                [equall(VARG1,nil),do(ARG3)])]
```

3) CLOSING INTERVAL - if two descriptions have different values for a linear descriptor, then we may combine them by suggesting that all values between those given are applicable. If we have descriptions for (ADD1 2) and (ADD1 9) we may propose the description applies for numbers between 2 and 9.

4) CLIMBING GENERALIZATION TREE - if an item in a descriptor is known to have a hierarchical ordering, and two descriptions have constants which share a parent node in that hierarchy, then we may "climb the tree" and suggest that the parent node is an appropriate descriptor. For example, if we have seen the function "explode" work on numbers and booleans, we could look at the parent datatype node to suggest that it works for all atoms (which would introduce alphanumeric atoms as well). This involves including some domain specific knowledge in the system. In the existing system the only generalization tree provided is that for datatypes.

5) CONSTANTS TO VARIABLES - The fewer the number of arbitrary constants in a description, the more general it will be. For example, a prediction that QUOTE returns its argument as its value is more general than a prediction that (QUOTE ALICE) returns ALICE as a value. This rule suggests that any constant may be replaced by a variable. In the pupil model this has been constrained slightly such that only constants which occur more than once may be turned into variables.

6) CONJUNCTION TO DISJUNCTION - If a description includes two descriptors joined by a conjunction, then a more general form would be to link those descriptors via a disjunction. For example, consider NIL which is both an atom and a list;

```
atom(nil),list(nil)  becomes
    or([atom(nil)],[list(nil)])
```

7) INDUCTIVE RESOLUTION - Given a condition P, such that when it is present with a condition C1, or absent while a condition C2 is present, then the event belongs to a particular class, we may reduce this to the statement that the presence of C1 or C2 indicates that the event belongs to that class. The following example illustrates this rule, but the result is not what we would expect;

```
american(X),lives(X,england) is an event "exile".
```

not american(X),lives(X,america) is an event "exile".

This results in;

or([lives(X,england)],[lives(X,america)]) is an event "exile".

2.-Deductive reasoning.

As has been described above, the model of the pupil relies on the presence of deductive reasoning methods which the pupil uses to make concrete predictions about the behaviour of the domain. It is only through these predictions that information about the state of the model can be obtained.

There has been a large amount of research on the nature of deductive reasoning, much of it appropriate for a computational model (e.g. see [Robinson 1979]). Unfortunately, time has not permitted the review of this literature within the current work. The lack of an adequately explored model of deductive reasoning is a major deficiency of this program design.

The process of deriving concrete predictions from the model of the domain is tackled by the problem-solving mechanisms (see above). In the absence of a suitable theory of deductive reasoning which could be used to model and develop the pupil's problem-solving skills, the system relies on its own problem-solver to determine predictions which the pupil is likely to make. This is not a satisfactory solution in the long-term, since the problem-solver is essentially a black-box expert, which should not be used as part of the student model; the end results are valid but the mechanisms for achieving those results are not like those of a human. This was discussed in chapter 5.

3.-Learning notes in the user model.

The mechanisms of learning which have been described up to this point provide the basis for a program which could learn LISP from observation. This is not the same as a model of our pupil, so we may ask what contribution the learning model makes to the pupil model.

For any given expression which the pupil tries out, the learning model can produce a set of possible hypotheses which the pupil could build about the behaviour of that expression. The set of possible hypotheses may be infinite, but for practical purposes we will assume that the learning model only generates a finite set. Each of these hypotheses can be used to provide predictions about other expressions in the language. The teacher may use these predictions to determine which hypothesis the student actually believes, and to generate counter-examples or confirmatory evidence as appropriate. Since this process of validation may take some time, IMPART keeps a note of the possible hypotheses which the pupil may be currently entertaining. In essence this forms the basis of the pupil model.

Since the hypotheses are essentially semantic descriptions of programming language statements, we may regard the set of hypotheses which the pupil holds at any given time as a definition of the language which the pupil thinks she is using. Because the semantics form a closed domain, one goal of the teacher is to make this language description equivalent to (not necessarily identical with) the correct definition of the language.

It is not generally possible to store all the hypotheses which a pupil could make throughout her history of interaction with the machine. It has been pointed out [Carbonell 1983] that a set of hypotheses is defined by giving the maximally-specific and minimally-specific elements of that set. All others can be generated from this.

In our case, we represent each interaction step by providing a maximally-specific description (essentially a statement of what occurred) and the minimally-specific CORRECT description which the pupil could achieve (i.e. incorrect generalizations are disregarded). These provide sufficient information to enable the teacher to guide the pupil towards the best generalization, while retaining the possibility of discussing more idiosyncratic interpretations. For example,

<u>EVENT</u>:

```
(SETQ Z B)
    5
```

<u>MAXIMALLY SPECIFIC</u>:

```
[
 type(Z,atom),
 type(B,atom),
 change(Z,5),
 out(value,5)]
```

i.e. Z and B are atoms. (SETQ Z B) gives Z a value of 5 and returns a value 5.

<u>MINIMALLY SPECIFIC</u>:

```
[
 type(ARG1,atom),
 type(ARG2,atom),
```

```
do(ARG2),
in(value,VARG2),
type(VARG2,number),
change(ARG1,VARG2),
out(value,VARG2)]
```

i.e. SETQ takes 2 arguments which are atoms. It evaluates the first and gets that value. The value must be a number. It changes ARG1 to have a value which is a number, and returns that number as the value of the expression.

All the examples given so far assume that the pupil has no knowledge of the language. In general we will be dealing with compound statements of which parts are already well understood by the pupil (if they aren't then the system should help the pupil to subdivide the problem). This can be handled by permitting the learning system to use (and reference) other hypotheses which the pupil currently holds.

It is not assumed that the hypotheses which the system generates correspond directly to those of the pupil. Instead, they are regarded as upper and lower bounds to the hypotheses which the pupil has made. The teacher can attempt to "home in" on the actual beliefs of the pupil by looking for evidence which will enable the distance between the bounds to be reduced. For this reason the process may be described as "bounded user-modelling".

4.-Other knowledge about the pupil.

All the information discussed so far is derived from the activities of the pupil. The teacher has high-level goals and the degree to which those goals have been achieved also constitutes part of the model of the pupil. For example, if the tutor wants the pupil to understand variable binding, then she will have some model of the pupil's level of understanding of variables.

This sort of information is closely linked to the interaction between pupil and teacher. In the following chapter the nature of the teacher's goals will be discussed and the user modelling associated with them will also be mentioned. For the present, it is sufficient to say that procedures for assessing the pupils' knowledge of each goal and maintaining the pupil model are associated with these goals.

5.-Summary of chapter 11.

The idea of "bounded user modelling" has been introduced. A method for deriving such a model from events in the environment has been described. This involves the application of a machine-learning paradigm to these events.

Chapter 12.

Discourse control for tutoring.

We have now examined all the sources of information available to the teacher during the course of the educational experience. We may turn our attention to the methods by which these may be utilised to produce a useful, structured, teaching interaction. One major component of this task is the general one of examining the nature of any interaction. The second major component is outlining the "teaching strategies" which embody the application of the educational theory in this domain.

The basic problem confronting the interaction systems is to take a set of hypotheses about items to teach (from the learning model), some top-down domain specific goals (the curriculum), some rules about educational style, some rules about "good interaction", and to blend them into an integrated discourse between pupil and teacher which satisfies the needs of each.

This chapter commences with a brief summary of some work on discourse structure which is relevant to this system. This is followed by a description of the actual architecture of the discourse systems used in IMPART. The section concludes with an example of the way in which the mechanisms operate.

1.-Relevant research on structured discourse.

There are several areas which may be relevant to the design of a mechanism intended to support a structured educational interaction. Three areas will be examined in greater detail. The first is work from the Intelligent Teaching System literature itself, the second is Educational research into classroom interaction, and the third is psycholinguistic models of discourse.

1.1-Teaching System dialogues.

In WEST [Burton 1982] , the dialogue comprises canned speech produced in response to a particular action of the pupil. The domain knowledge is represented as ISSUES (which are general items to tutor) and EXAMPLES (which are concrete instances of issues). When the pupil carries out an action each ISSUE tests whether it has been mentioned, and if so assesses (by examining the student model) whether it should contribute to the system's response. If the result is affirmative then a canned message linked to the issue is presented to the pupil. To prevent boredom the system is provided with several canned messages for each issue, from which one is selected on a random basis. Burton points out that this aspect of the system was not a major focus of research and identifies various useful modifications, such as providing a facility to combine messages from issues into a single succinct comment. It should be noted, however, that the system is essentially reacting to input on an event by event basis with no long-term interaction structure, and no facility for handling direct queries from the pupil.

GUIDON [Clancey 1979] attempts to support a mixed initiative dialogue with the student. In this it has dialogue goals, which are basically the same as the goals of a problem-solver tackling the case which GUIDON is attempting to tutor. The system can only attempt to satisfy one goal at a time, but it can maintain a sub-goal hierarchy. The top goal may be selected by the pupil but, rather than following the pupils goal structure, the system thereafter uses the sub-goal tree provided by MYCIN. Clancey identifies three knowledge sources which contribute to the interaction;

1) Conversational Knowledge.

The problems here are with general methods for directing and focussing the interaction. GUIDON uses "action patterns" which embody methods of achieving goals in a conversation. An equivalent structure for comprehending utterances is the "interpretation pattern". This follows work by Faught [Faught 1977] .

2) Domain knowledge.

A hierarchical structure is imposed upon the domain knowledge and this affects the decisions about relevance of particular items to the current discussion.

3) Knowledge of the communication situation.

In effect this models the dynamic aspects of the interaction which provide the context in which conversational knowledge can be applied. There are three major components;

i) Overlay model of student.

ii) Case syllabus - this is a list of important topics to discuss during a particular case. It is built into the system by the designer.

iii) focus record - this provides a simple representation of topics which were recently discussed.

The system operates by invoking a discourse procedure whose purpose is to select those domain-rules which may be relevant to the current discussion. These rules are then passed to the tutoring strategies which decide which rules to follow up and how to present them to the pupil (essentially choosing the details of content and style). The tutoring strategies invoke "discourse patterns" to present information to the pupil.

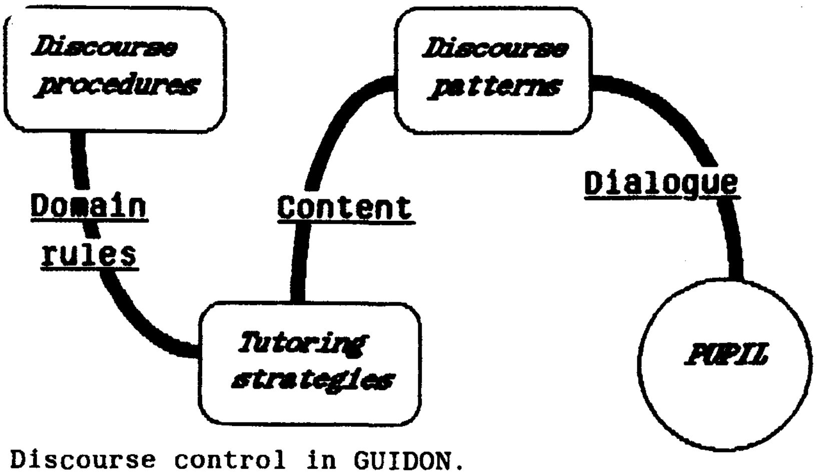


Figure 23 - Discourse control in GUIDON.

A difficulty with the design of GUIDON is that the interaction tasks seem to be divided between discourse procedures, domain rules, tutoring strategies and discourse patterns in a rather adhoc way.

In designing SCHOLAR, Collins attempted to produce a structured teaching interaction. In particular he pointed out that the interaction should build on previous knowledge of the student, should direct the response to student errors and support interactive diagnosis of problems. In developing the system, Collins took a set of preconceptions about the sort of discourse which his system could support and used this as a framework to analyse the teaching methods of several human teachers. The work focuses on Socratic learning methods.

The issues which Collins felt were of central importance were; topic selection, blending questions and presentation of material together, questioning on basic concepts, reviewing material, hinting and responding to errors. In this and subsequent work, a theory of teaching interactions has evolved which contains four main components;

1) Goals and sub-goals of an effective teacher. These are associated with presenting domain knowledge to the pupil and ensuring that she has techniques appropriate to use it. A taxonomy of these goals from [Collins 1982] is shown in Figure 24. At any given time the teacher is only satisfying one goal, but there may be others pending. In a given domain these rules are applied to domain specific knowledge.

2) Strategies to realize these goals.

3) Control structure to select and pursue various goals. If the system has multiple goals there must be some way of deciding which should be satisfied next. In another paper [Collins 1980] Collins subdivides this aspect into four components;

i) Strategies for choosing cases. In this terminology a case is a particular example to illustrate a point. These strategies identify examples which are appropriate to a given point.

ii) Student model. In Scholar this is more a model of the interaction than of the student. A note is kept of things which have been said and questions which have been correctly answered. Collins does not explore the nature of this model in great detail.

iii) Agenda. This is a list of the current goals and subgoals of the tutor. It is kept as a pushdown stack of topics. It is here that the clearest link to the interaction structure is seen. The current goal is the most recent thing on the stack. When the system decides that enough has been said about this goal, the system pops this goal and reverts to discussing the previous goal.

iv) Priority rules. The goals on the agenda are not added as they crop up in the interaction. Instead each goal is assessed for its importance and assigned a place on the agenda linked to that importance. The length of time assigned to discussing a particular topic is also determined by these rules.

Goals and subgoals of teachers.

1) Learn a general rule or theory.
   - 1a) Debug incorrect hypotheses.
   - 1b) Learn how to make predictions in novel cases.

2) Learn how to derive a general rule or theory.
   - 2a) Learn what question to ask.
   - 2b) Learn what is the nature of a theory.
   - 2c) Learn how to test a rule or theory.
   - 2d) Learn to verbalize and defend rules or theories.

Figure 24 - Tutoring rules in WHY.

1.2-Educational interaction research.

In general, educational research on classroom interaction has focussed on analysis of general features of the interaction. Various scales are used to sample the type of discussion which is taking place at short intervals. This allows categorization of general teaching styles (see e.g. [Bennett 1976]), but does not make any contribution to understanding the decisions taken with regard to details of topic presentation. Direct classroom observation is important for understanding teaching methods, but the current methodology is not appropriate for embodying such information in the precise way required by a tutoring system.

1.3-Psycholinguistic research.

The analysis of conversation structure is a fairly recent field within pscholinguistics. It is more difficult to approach than problems of syntax and parsing within an utterance since a conversation includes large amounts of domain-specific knowledge which make it difficult to abstract "general features of conversation". Some work has been done [Robinson 1982] on extending strict syntactic rules to structures larger than a single utterance, but in general these techniques have not been shown to transfer adequately.

More fruitful approaches to conversational analysis involve dividing the interaction into units which have a specific purpose, linked to the goals of the speaker or hearer, or a division based on changes in topic.

1.3.1-Dialogue games.

Levin and Moore [Levin 1980] looked at conversation with a view to dividing an interaction into a set of primitive units. This work begins from observations such as "a question is often followed by an answer". A number of these "adjacency pairs" have been identified [Schegloff 1973] . Power [Power 1979] pointed out that an interesting feature of these pairs is that they need not be adjacent. Consider the following conversation;

1 a: How many sugars do you want?

2 b: Is it tea or coffee?

3 a: tea.

4 b: One please.

In this, utterances 1 and 4 constitute an adjacency pair, though they are not adjacent. The intervening material is relevant to establishing the response. Levin and Moore extended this idea by analysing a dialogue into multisentence units according to the function which they are intended to fulfill. The function of a unit is related to a goal of the speaker. They call these units "dialogue games" and describe them as "frequently recurring established patterns of interaction which span several turns in a dialogue". From an analysis of a number of conversations via teletype between users of a large computer, they produce a set of common "games" (these are not intended to be a complete taxonomy).

1) Helping (person 1 wants to solve a problem, interacts with person 2 to reach a solution).

2) Action-seeking (person 1 wants some action performed and interacts with person 2 to get him to do it)

3)Information-seeking (Person 1 wants to know some specific information and interacts with person 2 to learn it)

4) Information-probing (Person 1 wants to find out if person 2 has some particular information, and interacts with him to find out)

5) Instructing (Person 1 wants person 2 to know some information and interacts with him to impart it)

6) Griping (Person 1 is unhappy about some state of affairs and interacts with person 2 to convey that unhappiness).

It is worth noting that these games are essentially defined in terms of the goals of the participants. Some of these are more obviously relevant to an educational interaction than others (e.g. 3,4,5) and it seems likely that there exist more specific games which directly embody educational goals.

A dialogue game is defined in terms of three sub-units; some parameters, some specifications and some components.

PARAMETERS represent the information other than function which must be available to a particular game. In the games listed above Levin and Moore found that only three parameters were required; two ROLES to identify the speaker and hearer in a particular interaction, and a TOPIC.

SPECIFICATIONS are restrictions upon the parameters which determine whether the state of the world is appropriate for the initiation of a particular game. For example, in the HELPING game, our specification would be that the person requesting help wants to achieve something, is unable to achieve that thing, and is permitted to achieve that thing, while the Helper must be willing and able to facilitate the person needing help in achieving that thing.

COMPONENTS control the actual detail of the game. They may be regarded as sub-goals towards achieving the goal defined by a particular game. In the case of helping they could be: 1) explain what was expected

2) describe what happened 3) helper offers explanation. These components may be simple utterances or they may invoke another game in order to achieve their goal.

The following is an example of the game for seeking information.

Game: INFO-SEEK

Parameters: SEEKER SOURCE INFO

Specifications:

SEEKER doesn't know INFO

SOURCE knows INFO

SEEKER wants to know INFO

SOURCE is willing for SEEKER to know INFO

Components:

SOURCE wants SEEKER aware of INFO

SEEKER wants SOURCE aware that

SEEKER is obligated to SOURCE

The mechanism for dialogue proposed by Levin and Moore regards the game-description above as a knowledge structure which may be acted upon by three cooperating processes to produce a dialogue. These processes are concerned with initiating a game, comprehending and producing utterances within a game, and terminating a game, respectively. These combine to produce a five step model of the discourse process;

NOMINATION - This is the process of activation a particular game. A toy version could simply announce that it intends to embark upon game X (Power's system does this, for example), but in real conversation this process is often implicit. Attempting to establish the parameters may result in nominating a particular game. In the system provided by Levin and Moore there are two detailed mechanisms for nominating games - one by a "spreading activation" model which brings games related to currently active concepts into focus, and one rule-based transformation of the input intended to propose goals for the speaker which can be matched against game-specifications.

RECOGNITION - The described mechanism includes a representation of Long-term and Working memory. The recognition phase involves resolving any conflicts produced at the nomination stage by referring to supporting evidence in memory.

INSTANTIATION - Once a particular game has been established it may be the case that some of its parameter specifications do not have corresponding entries in memory. For example, if someone invokes the info-seek game by saying "Will you tell me the height of the Eiffel Tower?", we can infer that the speaker wants this information, but does not have it. The instantiation phase simply takes inferences which can be made from the parameter specifications and notes them in working memory. In this way, even if the game fails to achieve its goal, we have updated our model of the goals of the speaker.

CONDUCT - Conducting the game corresponds to carrying out each of the components. These are built in to the game in such a way as to satisfy the ultimate goal of the game. There is typically a temporal ordering to these components.

TERMINATION - Termination may happen in several ways. A game may achieve its goal and exit conventionally or the other participant may interrupt. As soon as a parameter specification ceases to hold the game is deemed to be no longer relevant (e.g. If someone wants you to help her achieve a task, and that task is achieved or she decides that she no longer wants to achieve it, then there is no need to continue helping her to achieve it.

The goal-oriented nature of dialogue games is very important and distinguishes them from most other approaches to describing natural language. Levin and Moore explain that;

" Each Dialogue game can be seen as a problem-solving operator selected to accomplish some given high-level goal and then specifying a set of subgoals to pursue."

Levin and Moore list some generalizations about language which follow from their theory. It is worth noting how many of these refer to goals of the participants in the interaction.

1)Part of the comprehension of any utterance is to associate particular functions with it, inferring that the speaker is using the utterance as a means to accomplish one or more particular identified goals which he holds.

2) The speaker ordinarily holds multiple goals, and these are related in highly constrained ways.

3) The goals held by the two participants of a dialogue are not independent but rather are closely related in ways which strongly and systematically constrain co-occurence of goals.

4) These related sets of participants goals underly a significant amount of dialogue behaviour and the knowledge of these recurrent goal patterns is essential for language comprehension.

5) People use their knowledge of goal structures in dialogue to effect implicit communication of various kinds, including the performance of indirect speech acts and the implicit communication of assumptions about each other.

6) Changes of topic in dialogue are directly dependant upon changes in the participants goal structures, and are accomplished as side effects of goal structure changes.

7) Indirect communication, including indirect questions and requests, arises out of the part of language comprehension which associates

functions with utterances.

1.3.2-Goal-directed dialogues.

Richard Power has taken a similar approach to Levin and Moore in attempting to impose a structure on conversation. He produced a program in which two robots with different goals can converse in order to achieve those goals. The robots exist in the world described in chapter 7.

The major difference between Power's work and that of Levin and Moore is that since it is a model of conversation generation it must make some statements about the way in which an agent develops a set of goals and how these goals are linked to an interaction. As Power says "... there has been a good deal of work on language understanding and goal-directed behaviour; but the two have not got together." The robots have goals of achieving states in the world and they operate by building a planning tree to achieve those goals. In this program the robots communicate to keep each other informed about the state of their plan and their beliefs about the world. The robots may also ask each other to perform tasks.

Adding planning to the discourse involves Power in extending the idea of "discourse games" (which he calls "conversational procedures") such that they contain statements about the expected course which a particular game will follow, and the way in which the utterances are linked to the goal of that game. While Levin and Moore take it for granted that the components of a game will satisfy the goal of that game, Power attempts to justify every utterance within a game in terms of its relevance to the current state of the speakers plan. Power also introduces "planning procedures" which correspond to private thoughts. Planning procedures and conversational procedures may call each other in a reasonably unrestricted fashion. In this way a blend of reasoning and discussion is achieved.

The following example of a conversation procedure from Power's program should be compared with INFO-SEEK above, which is designed to achieve the same thing. The parameters are the same in both cases, comprising two ROLES and a TOPIC, but the body of Power's procedure appears to be less completely formalised.

Game: ASK

Parameters: SPEAKER-ONE,SPEAKER-TWO,STATE-OF-AFFAIRS

Actions:

1) SPEAKER-ONE composes a SENTENCE which expresses STATE-OF-AFFAIRS as a QUESTION, and utters it.

2) SPEAKER-TWO receives the SENTENCE and determines the STATE-OF-AFFAIRS to which it refers. He records that SPEAKER-ONE cannot see the object mentioned in STATE-OF-AFFAIRS, and then inspects his world model to see if STATE-OF-AFFAIRS is true. If he finds no information there he says I DON'T KNOW, otherwise YES or NO as appropriate.

3) SPEAKER-ONE reads SPEAKER-TWO's reply. If it is YES or NO he updates his world model appropriately. If it is I DON'T KNOW he records that SPEAKER-TWO cannot see the object mentioned in STATE-OF-AFFAIRS.

The sort of interaction which this unit would generate is as follows;

S1: Are you inside?

S2: No.

It is important to note that apart from specifying the content of this trivial interaction, the procedure also describes its effect on both hearer and learner. When both participants are using the same game it not only provides a framework within which to generate utterances, but also a context within which to understand them.

Power criticises his program for its lack of flexibility in dialogue. If something unexpected happens within a game, then that game is completely abandoned. Levin and Moore probably proceeded further on this aspect of the problem with the INSTANTIATION step which keeps provides a general method for keeping track of assumptions in dialogue. Power attributes this limitation of the program to the weakness of his representation of the current dialogue state - many of the utterances are understood by his system simply in terms of the previous remark; they are not related to a wider context.

In order to rectify this, Power proposes that a dialogue state should be represented by a structure which grows from the main goal. Every utterance should be comprehended in terms of its effect upon this structure. Power suggests the following necessary components for this representation;

- Assertions
- Goals
- Plans
- Candidate plans
- Goal-directed procedures
- Inference rules
- Justifications of;
    - Assertions
    - Goals
    - Plans

1.3.3-Dialogue context.

Rather than divide a conversation into units based upon function, Reichmann [Reichmann 1978] looked for topic-linked structures in recorded conversations. The basic supposition behind this work is that a conversation is "a structured entity whose utterances can be parsed into hierarchically related context spaces". These "context spaces" are groups of utterances which refer to a single issue or event. The analysis builds upon earlier work by Grosz [Grosz 1979] on task-related dialogue, which showed that the a task-related dialogue has an underlying structure that parallels the structure of the task being discussed. Grosz groups utterances in terms of the concept of a "focus space". Any explicitly mentioned item is in "explicit focus", while items linked to explicitly focussed items are brought into "implicit focus". During a conversation there may be many focus spaces in operation. A distinction is drawn between "active" focus spaces, which represent the current focus of conversation, "open" focus spaces, which represent focuses which are not current but which may again become current, and "closed" focus spaces which have been discussed and will not become active again.

Reichmann analyses an interaction into a sequence of "context spaces" together with some formal relationships between them. There are two types of context space, "issue context spaces" and "event context spaces". Issues refer to generalized activities while Events refer to particular instances (c.f. Issues and Examples as used in WEST (see above)). It is suggested that "The underlying structure of a conversation is the set of relationships that hold among its constituent context spaces".

An issue context space concerns;

1) A general issue of concern (the topic).
2) The actors and objects participating in the issue (if any).
3) The time of occurence of the issue (if any)
4) The duration period of the issue (if any)
5) Focus level assignments to each of the above.

6) State of the context space at a given time in the conversation.

An event context space concerns;

1) A particular episode and the events that occurred therein.

2) The actors and objects participating in the episode (if any).

3) The location at which the episode occurred.

4) The time of occurence of the episode.

5) The duration period of the episode.

6) Focus level assignments to each of the above.

7) A topic of, or point being expressed by, the event context space.

8) State of the context space at a given time in the conversation.

The content of these "context spaces" seems worryingly adhoc. They were derived to provide a formalism to represent each context which arose in two specific dialogues which were examined in detail. For the moment we will overlook this point and consider two examples from the taxonomy of relationships between contexts which Reichmann produced.

ILLUSTRATIVE AND RESTATEMENT RELATIONS.

This represents the case where some general issue is discussed, one or more specific instances on that issue are introduced, and then the original issue is restated. Consider an example;

" If you evaluate a LISP variable you get back the value associated with that variable {ISSUE}. When you evaluated A you got the value APPLE {EVENT 1}, and when you evaluated B you got the value BANANA {EVENT2}. These are the values associated with these variables {restatement of ISSUE}."

In this case our Issue is values of variables. This arises in the first sentence, and is restated in the final sentence. Between these are references to two events - evaluating A and evaluating B - which illustrate the general issue. In general this relation has the form;

State an ISSUE.

State one or more EVENTS.

Restate the original ISSUE.

The illustrative relation is subdivided into "reference illustrative" (which occurs when the hearer already knows the event context space) and "full illustrative" (when the hearer has no knowledge of the event context space and the speaker must give a full description of it.

GENERALIZATION RELATION.

A generalization relation is one in which a particular instance of something is discussed, and this is followed by a statement of an Issue (presumably some general case which follows from the specific case). For example,

" Before taking the first element of it, CAR evaluates its argument {EVENT}. Most Lisp functions evaluate their arguments before doing anything to them {ISSUE}."

State an EVENT.

State an ISSUE.

The major issues which Reichmann identifies in producing a coherent conversation are topic and focus.

An overall goal of a listener is to maintain a model of the current topic which adequately explains all utterances of the speaker. This involves assessing the relevance of each utterance to the current topic and either integrating it within that context or switching context to provide a new topic with which the utterance is compatible. Reichmann attempts to formalize this behaviour by distinguishing five types of "state" which a context space may be in, and providing some "Semantic relational rules" for assigning these states to particular spaces and guiding transformation between these spaces.

Focus is used to assign importance to individual entities within a conversation. It is not independent of topic; Choice of focus may affect topic and vice-versa. Reichmann represents this with a "focus level" assigned to each entity by "focus assignment rules". It is pointed out that these focuses are subjective and it is possible for the speaker and listener to have different focuses within the same interaction.

The theory outlined above provides an adequate framework for representing the dialogues studied by Reichmann. It is not clear how much extension would be necessary to deal with other instances of dialogue. Another point to bear in mind is that this is essentially a descriptive theory - it offers no method for linking these structures to the underlying intentions of speaker and hearer. Finally, we see throughout this work a central position given to the ISSUES/EVENTS distinction. This seems to be a specific case of a generalization-specialization relationship. Replacing issues and events with a spectrum of "generality levels" would not adversely affect the theory. It is not immediately clear that this is the only dimension controlling topic changes in conversation.

2.-Architecture of a discourse system.

In a teaching situation there are two major driving forces controlling the interaction: the goals of the teacher and the state of the pupil. The architecture of the discourse system attempts to reflect various aspects of real discourse. In this section we will describe the features which we are attempting to model, then give details of the actual mechanism used in IMPART.

2.1-Features to model.

2.1.1-Opportunistic tutoring.

One aspect of interaction control is that which is external to the teacher. This involves direct statements by the pupil and situations which arise as the result of actions of the pupil. Such occurences interact with the teacher's plans. A poor teacher will override this part of the educational interaction, ignoring the opportunities which are offered. In some sense this is no better than providing the pupil with a passive aid such as a text book. A good teacher should make use of these events, incorporating them into her plans to produce a richer interaction. This type of opportunistic tutoring is characteristic of Intelligent teaching.

A major component of this aspect of interaction is the set of goals held by the pupil. It is important to try and deduce what these goals are. These goals alone do not provide a sufficient representation of the factors external to the teacher. The behaviour of the environment often differs from the pupils expectations. This must be monitored since many useful tutoring steps can be taken at precisely those points where the pupils expectations differ from the behaviour of the environment.

The application of a theory of learning to observations which the pupil can make provides all the basic information for opportunistic tutoring based on events which occur in the environment. As was shown in the previous section, this theory produces a set of possible hypotheses which the pupil could hold, which in turn indicate a set of possible items to tutor in any given situation. Deciding what to actually tutor and how to teach it involves considering the relevance of each item to the current state of the interaction.

It is very valuable to provide the pupil with the facility to interact directly with the teacher by asking questions. This encourages the pupil to take control of the interaction and supplies a valuable feedback mechanism which allows the teacher to discover which features of a problem are actually worrying her pupil.

2.1.2-Goals of the teacher.

The teacher knows various abstractions which make the language easier to understand. Where an explanation is necessary the teacher should couch it in terms of these concepts. A decision must be taken about the relevance of a particular discussion to the current interaction, and this assessment is of major importance in deciding what to teach. In some sense tutoring these concepts corresponds to a goal of the teacher. These concepts may be regarded as an embodiment of the curriculum.

Let us consider how the goals of the teacher affect the interaction. If a teacher wishes the pupil to understand the origins of coal she will plan a method of presenting this information to the student. This might be thought of as part of the long-term structure of the interaction. This corresponds to a type of goal which is not in general satisfiable, but which may be reduced in importance by repeated action. As an example of this, if we wish a pupil to understand about integration of mathematical functions, we cannot expect to offer this information once and expect it, and all its consequences, to be understood. Satisfying such a goal may begin in junior school with counting squares under a graph to determine the area. This can gradually be extended via methods of approximation until we are able to present a mathematical proof of the method of limits. Even now it may be many years before the pupil fully understands integration - and that goal may never be completely satisfied. Each of these stages may be regarded as an attempt to satisfy the goal of teaching integration. They use very different methods, and none can be said to have finally satisfied the goal. This is reminiscent of the earlier discussion of when to teach things, which suggested that there is an appropriate way to teach anything to a pupil at any stage of development.

It is important to try and represent the domain specific knowledge in such a way that it can be presented to the pupil and used to guide the course of the interaction. It is also important to keep this domain knowledge separate from any general mechanism used to control the conversation, although presentation methods will depend upon the knowledge being taught.

2.1.3-General discourse features.

Examining the work reviewed above, we find that attempts have been made to structure a dialogue by dividing it into units selected with respect to content (e.g. SCHOLAR, Reichmann) or with respect to function (e.g. Power). It seems likely that the structure of any real dialogue wil depend upon both these issues. It is also worth noting that a major division of content used by most researchers is the abstractness of the material being taught. WEST uses concrete INSTANCES to illustrate abstract ISSUES, Reichmann uses concrete EVENTS to illustrate abstract ISSUES and so on. I would propose that the binary division of subject-matter into concrete and abstract units is artificial. Concreteness is a relative property of pieces of subject matter. I wish to suggest that smooth transitions along a dimension of concreteness are of great importance in maintaining smooth discourse structure.

Another major feature of discourse which should be modelled is the idea of Conversational Focus. This term is used to describe the way in which certain pieces of subject matter may be more relevant to the current discussion than others at a given time. The set of things which are currently "in focus" affect the interpretation of utterances.

2.2-Actual discourse architecture.

In IMPART the dialogue will be structured by the combination of high-level components (corresponding to goals of the teacher) and low-level components (corresponding to events external to the teacher). The resulting dialogue should be appropriate to the current situation, and should achieve smooth transitions in topic and focus of interaction.

There are four major constituents of the discourse system; a "conversational context", a set of "topic controllers" which can invoke "discourse games", a "bidding mechanism" and a high level "conversational controller". Each of these will be described separately.

2.2.1-Conversational context.

Continuity and consistency within the interaction is maintained by introducing the idea of a "conversational context". As in the work of Grosz [Grosz 1979] , this corresponds to a set of items which are implicitly or explicitly at the focus of the current conversation. Items which have been brought into focus decay in importance over time unless they are mentioned again.

The context contains three major components. The first of these has the obvious function of noting what has been recently said. This is done by associating a focus rating with each topic controller (see below). The pupil model also acts by affecting the conversational focus, since the set of possible hypotheses which the pupil may retain bring topics associated with those hypotheses into focus. The third context component, currently unimplemented in IMPART, should be the set of items referred to in pupil initiated interaction steps.

2.2.2-Topic Controllers.

The approach to representing domain-specific knowledge which will be taken here is to attempt to rationalise the idea of "issues" introduced in the WEST system. These issues are useful ways to group knowledge according to topic, but they are essentially terminal items which can simply present themselves to the pupil as canned pieces of text. Advice from WEST consists of a sequence of terminal items.

We will introduce the idea of a " topic controller " which contains a chunk of domain-specific knowledge. Instead of being a terminal piece of text, each Topic Controller is an active item with associated declarative and procedural knowledge, capable of controlling a dialogue in its specialist area. A controller may be invoked when its specialist subject appears, and it may be invoked with any one of a variety of goals to achieve. The controller may produce interactions with the pupil and it may call other controllers to talk about dependant issues within that interaction. A controller may also invoke games to achieve specific goals, providing those games with domain-specific information. As well as providing a means for generating interaction, the topic controller also provides a context within which interaction may be understood. When a controller has achieved its current goal, or decisively failed to do so, control will be returned to the unit which called it.

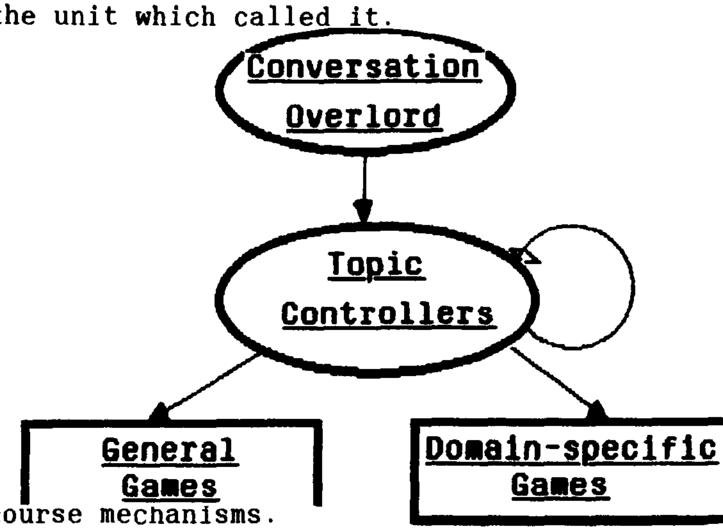


Figure 25 - Structure of discourse mechanisms.

This "topic controller" based structure may be seen to support many of the dialogue features which are being modelled.

A topic controller is essentially a topic-based unit of interaction, but within that unit there may be differing goals to be achieved which correspond to function-based interaction units. This supports both forms of discourse analysis described above. For example, context changing rules [Reichmann 1978] become rules for changing topic controllers, while functional units are represented by the set of discourse games which may be called in a particular context.

The actual dialogue games used are essentially the same as those of Richard Power [Power 1978] .

1: Info-seek. Attempt to get information from the other participant in the dialogue.
2: Info-probe. Check whether the other participant has some information of which you are already aware.
3: Illustrate. Give a more concrete discussion of some topic.
4: Impart-info. Offer information to the other participant.
5: Request-action. Try to get the other participant to do something.

The major difference is the addition of a game called "illustrate". This provides a mechanism for increasing or decreasing the concreteness of the discussion. In this way it removes the need for a distinction between events and issues which was mentioned earlier.

As was mentioned above, the mechanism keeps some collection of topics " in focus" during the conversation. There are two mechanisms involved here. The first is a RECENCY TAG associated with each controller which maintains the implicit focussing described by Grosz. A second mechanism is the subconcept ordering imposed upon the controllers. Since one controller may invoke another (or recursively invoke the entire conversation mechanism), there will be a hierarchy of controllers in action at any given time. All these controllers are in conversational focus, either as active context spaces or as open context spaces. This mechanism corresponds to a push-down stack model of conversational focus. It was mentioned above that this is not adequate to describe all the features of topic change found in human conversation, but it accounts for many of those features.

Apart from sectioning the domain knowledge, it may be observed that a topic controller or conversational controller need not know how a unit which it invokes achieves the goal which it was given. This allows the contollers to blend linguistic and non-linguistic methods within an interaction.

2.2.3-Bidding mechanisms.

The mechanisms which govern the conversation are clearly domain independent, but involve decisions on issues such as the relative importance of topics which require some domain knowledge. This is reflected in this system by allowing the control strategies to take decisions based upon simple numeric "bids" which are generated by topic controllers. In this way the domain knowledge is kept separate from general control strategies.

The bid made by a controller is based upon several components;

1) Intrinsic importance. A measure of the importance of each topic is built into the curriculum. If all other things are equal, the topic which is talked about is that which was considered most important by the curriculum designer.

2) Extrinsic importance. The relevance of the topic with respect to the current conversational context is assessed. This involves combining the RECENCY TAG for the topic controller with an assessment of its relevance to the current set of hypotheses which the pupil could hold.

The RECENCY TAG is simply a number indicating the last interaction step on which the controller was invoked. The larger the difference between this and the current step tag, the more time has elapsed since the controller was invoked. In essence we may consider that the controller gradually decays in relevance as time passes since it was mentioned. Given that this is the case, there are several possible ways in which the bid could depend on this tag; we might try to keep issues in focus by restating them, or we could adopt the strategy of focussing on those items which have dropped from view. It seems likely that a real conversation involves a complex dependence on this parameter. The RELEVANCE is assessed by examining the set of hypotheses produced by the learning model. If the TRIGGER associated with a particular component is present, then the bid carries greater weight. The RELEVANCE may be 1 or 0 depending on whether the trigger is present or absent.

A more sophisticated form of bidding would involve similar examinations for sub-concepts of a particular topic.

3) Level talked and tested. Within a particular topic controller, there is a set of levels of complexity at which the controller can talk. Associated with each of these is a set of procedures intended to test the pupils understanding at that level. It is clear that a teacher should attempt to increase the level of understanding of her pupil, but that progress to a more complex level should not be made unless the current level is well understood. For this reason, the bid includes a component which is based on both the difference between the current level and the maximum level of complexity, and the difference between level talked and level assessed. The bid is proportional to the difference between current level and maximum level. The link to level tested is more complex, since we do not wish to introduce new material if the current level is untested, but we would wish to carry out tests. This means that the bid should be linked to the goal which the controller will attempt to satisfy. More will be said about this later.

4) Sub-concepts. The topics are ordered according to a useful precondition structure. The length of time for which a topic will hold the conversation is partially determined by the number of sub-concepts which it calls. This information may also be part of the bid.

The actual dependence on these factors which the system uses is somewhat adhoc, being based on personal views about the effect of these factors. The following bidding function is applied;

BID= INTRINSIC*EXTRINSIC*LEVEL*SUBCONCEPTS

where EXTRINSIC = (CURRENT STEP - RECENCY TAG)(1+TRIGGER)

LEVEL = (TESTED+1/TALKED+1)*(MAXLEVEL - TESTED)

It is clear that the importance of a topic is in some sense directly proportional to the components mentioned above. Some clarification should be made, however, of the nature of the EXTRINSIC and LEVEL components.

Since EXTRINSIC is proportional to the difference between the CURRENT STEP and the RECENCY TAG, topics will increase in importance if they are not discussed for a long time. This ensures a wider discussion than would occur if the most recently mentioned things continued to be discussed. The TRIGGER factor causes a large change in the importance of the topic; this is reasonable since it is expected that things which are currently in context will be more relevant to the discussion than those which are not.

The TESTED/TALKED component of the LEVEL bid ensures that the system doesn't go far beyond the current known level of understanding of the pupil. Basically, the system cannot talk about something more complex unless it believes that the student has a reasonable grasp of the things which have already been discussed. The second component of the LEVEL ensures that new topics will be introduced, and ones which have been thoroughly discussed will gradually drop out of the conversation.

2.2.4-High level control.

The high level control strategies must blend the output of individual topic controllers into a smooth interaction. This involves preventing interference between controllers, and performing general functions such as marking topic boundaries. It is also claimed that a large component of the tutoring strategies operate at this domain-independent level.

The control structure consists of an algorithm for collecting bids from topic controllers and subjecting them to a filtering process governed by the tutoring rules. The Conversation Controller is then responsible for passing control to the appropriate topic controllers in the appropriate order.

2.2.4.1-A control algorithm.

1) Get bids from all issues.

2) Choose issues which may be given control.

3) Order and filter the selected issues, imposing any necessary presentation constraints.

4) Carry out an interaction step based on this ordering.

5) Go to number 1.

The first step involves collecting bids from each of the topic controllers. These bids should embody many of the fluctuating aspects of the interaction in a domain independent manner. They are numbers.

A subset of the bids will be chosen based on a simple mathematical selection rule. This step eliminates those topics which are not sufficiently in focus to discuss.

The topic controllers which have been chosen are examined by the tutoring strategies to decide which will actually be given control of the conversation. This may involve negotiation between controllers and tutoring knowledge in order to determine what specific goals the controller wishes to satisfy. The result of this stage is an ordered set of topics to discuss which may include constraints to be set upon each. At this point the conversation controller may also insert items of its own, such as " Thats enough about that, now lets talk about this", to delimit topic boundaries.

The course of action selected in the previous step is executed. Since this involves interaction with the pupil, unexpected things may happen. For this reason it is important to permit unconventional exits from the interaction. If control is unexpectedly returned to the conversational controller, it moves on to the next step in the algorithm.

The final step is to return to step 1, which means that an interaction is based upon an iteration of this algorithm.

It should be noted that this mechanism does not involve any domain-specific knowledge. It is also worth observing that the opportunistic components of the interaction are dealt with through the effect which conversational context has upon the overall bid.

2.2.4.2-Tutoring rules.

The task which the tutoring rules must complete is that, given a preliminary set of topics to be discussed, they should link the separate issues to produce an interaction plan (a structured set of goals), which is in accordance with the educational constraints upon the system. A set of rules adequate to describe a particular educational approach fully has not been developed, but it is hoped that the following examples will illustrate the flavour of these rules.

1) Linked precondition. If two topics share a subconcept, which would be invoked by each rule individually, then the subconcept should be invoked independently before the topics are discussed.

e.g. "Function-call" and "evaluation" have the joint sub-concept "value". Where they would produce i) below if treated separately, applying this rule would produce ii);

i) " This is a function call to QUOTE with argument FRED, which returns value FRED. LISP is described as evaluating something, when it executes a statement. Evaluation always produces a value, and sometimes has other effects as well."

ii) " In this example, FRED is the value of this expression. A value is returned when any Lisp expression is evaluated. Evaluating an expression involves determining the effects of a statement. The statement here is a function-call, in which a function (QUOTE) is applied to an argument (FRED)."

2) Avoid repetition. If several topics share the same functional goal (which may be discovered by a direct interaction between tutoring rules and domain controllers), then they should not appear consecutively. This may involve preventing one or more topics from being discussed. This rule will prevent interaction segments such as;

" What is the function-name in the above example? What is the value in the above example? What is the argument in the above example?"

3) Generalize wherever possible. If two or more topics appear which are a subconcepts of another topic, present the more general topic with references to the subconcepts. An example would be if "lists" and "atoms" are both possible topics. They are subconcepts of "datatype", and so the interaction should feature datatypes with reference to lists and atoms.

4) Hinting. If a topic has a small difference between level talked and level tested, and is not going to talk to a higher level, restrict the topic to an indirect statement. For example, if a pupil has been successfully giving lists to an APPEND statement, and then gives it an atom this will be "unexpected" in the sense that level talked and tested should be fairly similar. A simple "Did you mean to do that?" may well be sufficient to prompt the pupil to follow up this action.

Figure 26 - Sample interaction rules from IMPART.

It is proposed that a production-rule formalism is well suited to the task of representing these rules in the conversation controller. Such a representation would allow domain-dependent and domain-independent rules to be added separately and modified easily. Only a simple version of this controller has been implemented.

3.-An example.

To illustrate the mode of operation of the dialogue systems, this section outlines the way in which some topic controllers would interact to produce a structured discussion. Most of the internal details of the controllers are unprincipled and uninteresting, but the bidding and constraining process is worth examination. The following three tables show the states of three topic controllers (we will assume the rest are irrelevant);

****

Topic controller: EVALUATION

Trigger: (do ARG1)	Recency: 4
Level talked: 5	Level tested: 3
Intrinsic importance: 50	
Sub-topic controllers: VALUE FUNCTION-CALL ATOM	

****

Topic controller:VALUE

Trigger: (out $value$ ARG1)	Recency: 2
Level talked: 3	Level tested: 3
Intrinsic importance: 20	
Sub-topic controllers:()	

****

Topic controller: ATOM

Trigger: (type ARG1 $atom$)	Recency: 4
Level talked: 3	Level tested: 3
Intrinsic importance: 20	

Sub-topic controllers: ()

Suppose that the pupil has evaluated the expression "(QUOTE FRED)". The possible set of things to talk about in this instance (as shown in chapter 11), include functions, values, datatypes and evaluation. If it is assumed that the current interaction step is number 5, then the three topic controllers shown above would bid as follows;

evaluation: 50.(5-4)(1+0).(4/6)(10-3).3 = 630

value: 20.(5-2)(1+1).(4/4)(8-3).1 = 600

atom: 20.(5-4).(1+1).(4/4)(8-3).1 = 200

These bids would produce an ordering of these topics according to current importance. Filtering this list would eliminate "atom" as being insufficiently relevant. The resulting pair of topics is passed to the tutoring rules. The only rule which fires is that which identifies "value" as a subconcept of evaluation. In consequence, the list is replaced by an item indicating that "evaluation" should be given control of the conversation, with a goal of describing the subtopic of "value".

If "atom" had not been mentioned since step 1, then its bid would have been 800, and hence it would have been the major conversational topic. Experimenting with the bids clearly demonstrates the way in which the topic structure is dependent upon the current conversational context.

The mechanisms described here are still at a very rudimentary level. In particular the means of linking these topic choices to natural language generation has not been explored. This presents an interesting area for further research.

4.-Summary of chapter 12.

A mechanism which is intended to support a structured interaction has been described. Its link to the pupil model has been discussed. Connections to current work on conversation structure have been outlined.

Summary of part 3.

An intelligent tutoring system to tutor programming languages has been described. A PROLOG representation for formal language semantics has been introduced. A programming environment has been described. A learner-based pupil model has been discussed. A mechanism for maintaining a structured interaction has been outlined.

Part 4 -

Conclusions.

Chapter 13.

Conclusions.

1.-Introduction.

The final part of the thesis will present an example of a tutoring interaction which the system can actually support. This should be compared with the hypothetical interactions which were given earlier. Following this, the major themes of the thesis will be summarised, the limitations of the tutoring system acknowledged, and some directions for further research proposed.

2.-An example interaction.

In this section we will show an example of an interaction with the system. A table will be used to show the state of the relevant topic controllers, the relevant conversation rules and the discourse games which are used. Visible steps in the interaction are numbered and placed in boxes for clarity. For simplicity, the sub-topic part of the bid has been omitted (it will always assumed to be 1). The bids depend primarily on variations in the "recency", "talked", "tested" and "trigger" components.

2.1-A new user.

A new pupil sits down at the system and indicates her presence by pressing a help key.

Current-Step 1: Initiate interaction, new user.

Name	Recency	Talked	Tested	Intrinsic	Maxlevels	Trigger	Bid
atom	0	0	0	20	5	0	100
list	0	0	0	25	5	0	125
sexpr	0	0	0	40	8	0	320
evaluation	0	0	0	50	10	0	500
argument	0	0	0	40	5	0	200
function	0	0	0	40	10	0	400
value	0	0	0	20	7	0	140
intro	0	0	0	600	1	1	600

Discourse-games: IMPART ACTION-SEEK
Conversation rules: none.

The presence of it's trigger situation (all other controllers have 0 for all parameters i.e. nothing has been done) causes the special topic controller "intro" to put in the largest bid. This presents a canned introduction to the pupil. Since it only has 1 level to talk at, the "intro" subsequently becomes inactive. The final action of the "intro" controller is to invoke an "action-seek" dialogue game which is responsible for the final sentence in the introduction. This game remains active until it is satisfied.

```
******************************************************************
*  1) TEACHER: Welcome to IMPART.  There are two menus,          *
* one with FUNCTIONS (things which carry out actions in lisp)    *
* and one with editor operations.   Use these menus to produce   *
* an EXPRESSION (something understood by LISP) on the screen, and *
* then EVALUATE (find out what lisp does with it) it.  I suggest *
* you start with the QUOTE function - try filling the slot with  *
* your name.                                                     *
******************************************************************
```

Current-Step 2: State after speech has been presented.

Name	Recency	Talked	Tested	Intrinsic	Maxlevels	Trigger	Bid
atom	0	0	0	20	5	0	200
list	0	0	0	25	5	0	250
sexpr	0	0	0	40	8	1	1280
evaluation	0	0	0	50	10	1	2000
argument	0	0	0	40	5	0	400
function	0	0	0	40	10	1	1600
value	0	0	0	20	7	0	280
intro	1	1	1	600	1	0	0

Discourse-games: none.
Conversation-rules: none.

The speech placed triggers in the conversational context for "sexpr" "evaluation" and "functions". "Evaluation" and "function" made large bids, but a tutoring rule indicates that no controllers should be allowed to control the interaction immediately after the initiation of an action-request (i.e. If you've asked the pupil to do something, give her time to have a go before interfering again). If the pupil did not respond to the request after a reasonable time, then the bids shown above would be used in deciding how IMPART should react.

```
*****************************************************************
*                                                               *
* 2)  PUPIL: enters (QUOTE FREDA) and evaluates it to get FREDA. *
*                                                               *
*****************************************************************
```

Current-Step 3: State after expression has been evaluated.

Name	Recency	Talked	Tested	Intrinsic	Maxlevels	Trigger	Bid
atom	0	0	0	20	5	1	600
list	0	0	0	25	5	1	750
sexpr	2	0	0	40	8	1	640
evaluation	2	0	0	50	10	1	1000
argument	0	0	0	40	5	1	1200
function	2	0	0	40	10	1	800
value	0	0	0	20	7	1	840
intro	1	1	1	600	1	0	0

Discourse-games: ILLUSTRATE IMPART ACTION-SEEK
Conversation-rules:
1} Remove topics with unmentioned preconditions.
2} Combine topic/sub-topic pairs into a single unit.
3} Order topics by size of bid.

The recency level have been affected for those topics whose triggers were present at step 2. The learning model has generated a hypothesis-pair;

quote(freda) [type(freda,atom),out($value$,freda)]

quote(ARG1) [type(ARG1,sexpr),out($value$,ARG1)]

which sets triggers for "atom", "function", "value", "argument" and "sexpr". The bids are collected and filtered, leaving "evaluation", "argument", "function", "sexpr" and "value" (as shown above). The conversation controller takes this set and applies its conversation rules to decide which should actually control the discourse. "Sexpr" is eliminated since it's trigger was set with a "climbing generalization tree" rule, and the preconditions for introducing this topic have not been satisfied. The conversation controller recognises "value" as a sub-topic of "evaluation", and "argument" as a sub-topic of "function". Hence it rewrites the structure to call each major topic in turn, constraining it to invoke the sub-topic. It orders the two major topics such that the one with the highest combined bid is mentioned first. The final list of controllers is

((FUNCTION ARGUMENT)(EVALUATION VALUE))

i.e. Invoke FUNCTION with a goal of talking about ARGUMENTS, and EVALUATION with a goal of talking about VALUES. If we insert the choice of dialogue games into this list, we get the following structure;

((FUNCTION ILLUSTRATE IMPART (ARGUMENT ILLUSTRATE ACTION-SEEK))

(EVALUATION IMPART (VALUE ILLUSTRATE))) There is a direct correspondence between this structure and the structure of the following speech;

```
**************************************************************************
*                                                                        *
*  3)  TEACHER:  In the example you have just produced, QUOTE is a lisp  *
* FUNCTION; that is, an action which can be performed upon some ARGUMENTS.*
* In this case FREDA is the argument.  Why not try changing the ARGUMENT  *
* to QUOTE and seeing what happens.  EVALUATING a FUNCTION means finding  *
* out what lisp does when it carries out that action.  In this case, QUOTE*
* returned a VALUE FREDA.                                                 *
* seeing what happens.                                                    *
**************************************************************************
```

<u>Current-Step 4: State after speech</u>.

Name	Recency	Talked	Tested	Intrinsic	Maxlevels	Trigger	Bid
atom	0	0	0	20	5	0	400
list	3	0	0	25	5	0	125
sexpr	3	0	0	40	8	0	320
evaluation	3	2	0	50	10	0	167
argument	3	1	0	40	5	0	100
function	3	2	0	40	10	0	133
value	3	1	0	20	7	0	70
intro	1	1	1	600	1	0	0

Discourse-games: none.
Conversation-rules: none.

All the triggers have been turned off, and the corresponding recency tags have been modified. All the bids are below the cutoff threshold, and we are again waiting for an action-seek game to be satisfied, so nothing is done by the teacher. The topic controllers which have just been active produce very low bids because they have "talked" without testing that their information has been understood.

```
***************************************************************
*  4)  PUPIL: replaces FREDA with JOE.  Evaluates (QUOTE JOE) *
* to get JOE.                                                 *
***************************************************************
```

Current-Step 5: State after evaluation of expression.

Name	Recency	Talked	Tested	Intrinsic	Maxlevels	Trigger	Bid
atom	0	0	0	20	5	1	1000
list	3	0	0	25	5	0	250
sexpr	3	0	0	40	8	1	1280
evaluation	3	2	0	50	10	0	333
argument	3	1	0	40	5	1	400
function	3	2	0	40	10	1	533
value	3	1	0	20	7	1	280
intro	1	1	1	600	1	0	0

Discourse-games: ILLUSTRATE IMPART
Conversation-rules:
1} Remove topics with unmentioned preconditions.

The learning model has produced the hypotheses;

quote(joe) [type(joe,atom),out($value$,joe)]

quote(ARG1) [type(ARG1,sexpr),out($value$,ARG1)]

which sets triggers for "atom", "function", "argument", "value" and "sexpr" as in the previous case. "Atom" and "sexpr" make large bids, but "sexpr" is eliminated because it's preconditions have still not been discussed. "Atom" gains control of the dialogue. It uses the dialogue-game sequence "illustrate", "illustrate", "impart".

```
*******************************************************************
*  5) TEACHER: JOE is an ATOM.  In the previous example, FREDA was *
* an ATOM.  An ATOM is a sequence of letters.                      *
*******************************************************************
```

Current-Step 6: State after speech.

Name	Recency	Talked	Tested	Intrinsic	Maxlevels	Trigger	Bid
atom	5	1	0	20	5	0	50
list	3	0	0	25	5	0	375
sexpr	5	0	0	40	8	0	640
evaluation	3	2	0	50	10	0	500
argument	5	1	0	40	5	0	100
function	5	2	0	40	10	0	133
value	5	1	0	20	7	0	70
intro	1	1	1	600	1	0	0

Discourse-games: none
Conversation-rules: none

After presenting this speech, all topic controllers are below the threshold level for bids, so nothing is done. If the pupil continued to work with the environment, or to query the teacher, then this would affect bids and may result in IMPART taking some action. Instead, let us assume that the pupil does nothing. After waiting for a reasonable length of time, IMPART forces a new interaction step.

Current-Step 7. Introducing a new topic.

Name	Recency	Talked	Tested	Intrinsic	Maxlevels	Trigger	Bid
atom	5	1	0	20	5	0	100
list	3	0	0	25	5	0	500
sexpr	5	0	0	40	8	0	1280
evaluation	3	2	0	50	10	0	667
argument	5	1	0	40	5	0	200
function	5	2	0	40	10	0	267
value	5	1	0	20	7	0	140
intro	1	1	1	600	1	0	0

Discourse-games: ILLUSTRATE IMPART ACTION-SEEK
Conversation-rules: none.

The strongest bid is from "sexpr". Its preconditions have now been (partially) satisfied, since atoms have been discussed. This topic controller gains exclusive control of the interaction. It invokes subconcepts "atom" and "list". The former with an "illustrate" goal, and the latter with an "impart/illustrate" goal (since it is a new topic).

```
****************************************************************
*  6)  ABC is an ATOM.  A LIST is a                            *
* sequence of ATOMS and LISTS enclosed in a pair of brackets. *
* (A B C) is an example of a list.  ATOMS and LISTS are       *
* both types of S-EXPRESSION.  Anything which can legally be  *
* a LISP expression is an S-EXPRESSION.  Try giving a LIST as *
* an argument to QUOTE.                                        *
****************************************************************
```

This section has given a brief view of the way in which IMPART maintains an interaction structure. It should be borne in mind that, while the mechanisms for moving between topic controllers (as described here), have been systematically designed and implemented, the internal workings of the topic controllers are still somewhat ad-hoc in nature. This area requires further research.

3.-Conclusions.

The overall intention of this thesis has been to outline a framework within which tutoring systems can be implemented. There has been a discussion of general background issues. The detailed design of a tutoring system for LISP has been discussed. This tutoring system is intended to provide a concrete instance of a design within the given framework.

IMPART has not been fully implemented, but a series of programs intended to demonstrate the feasibility of the system has been described. In conclusion, the main points which have been established will be re-iterated.

3.1-Computers and Education.

The computer provides a tool which enables us to radically rethink our views on the nature of education. It is suggested that the use of computers will require developments away from the current mainstream of educational thought. It is extremely important to ensure that any actual tutoring system which is produced is designed with due consideration given to the educational perspective into which it must fit. A system which does not emphasise these issues will never be suitable for use as an educational aid in a realistic setting.

It is proposed that it is necessary to think of computers in terms of educational theories based upon one-to-one interaction and individual tuition. In particular, it is suggested that the work of Jean-Jacques Rousseau forms a suitable basis from which to start investigating these issues.

3.2-Programming environments.

A syntax-directed editing environment aimed specifically at novices has been described and implemented. It has been observed that the constraints on such environments when intended for novices are very different from the constraints on syntax-directed editors for experts. A case has been made for the use of such environments as a first introduction to a new language.

3.3-Intelligent tutoring systems.

A generic structure for tutoring systems has been proposed. This is based upon the idea of guided discovery learning as the starting point for all educational interactions. It is suggested that this may suitably embrace all teaching styles, since the freedom of this method allows pupil or teacher to move towards a more constrained interaction if it is considered appropriate. This is in contrast to tutoring systems which begin with a particular constrained style of interaction, since they lack the flexibility to change to a more appropiate style.

IMPART is an attempt to move towards a "Learner-oriented" tutoring paradigm. It moves away from expert-based designs. The proposed tutoring structure specifies the way in which a model of the learning process, some domain-specific knowledge, and some conversational knowledge may be combined in a teaching interaction.

The system implemented here has been set up as a flexible framework for tutoring. None of the components (e.g. the learning model, the discourse mechanisms) are in a definitive form, but the design allows them to be developed independently and plugged together. Alternative versions of the various components should be developed and tested (for example, to see how other models of learning would affect the interaction). In this sense the system also provides a framework for exploring the nature of teaching.

3.4-User modelling.

A unique feature of IMPART is the use of "bounded user-modelling". By describing the way in which a model of the learning process can be used to assign bounds to the current knowledge state of the pupil, the system offers a genuine alternative to the "expert-based" approaches of subset and perturbation modelling.

The application of current machine-learning methods to the user-modelling tasks has demonstrated the way in which "bounded user-modelling" can operate without a complete, psychologically valid model of the learning process.

3.5-Formal semantics.

A formal semantics for programming languages has been described in detail. This semantics is directly representable as PROLOG clauses, and consequently is eminently suited to automated reasoning about programs.

A language interpreter, trace package, and problem-solver based around this formalism have been described. These tools have not been fully developed, and each requires further exploration. In particular, the use of an annotated semantics as the basis for problem-solving deserves further investigation. It is proposed that this is an important part of developing a "glass-box" deductive reasoning mechanism which can be shown to the pupil.

The semantics is not yet complete. There are some language features which it is unable to handle, and it is less pure mathematically than could usefully be. It should be further developed, preferably to the point at which its equivalence with other semantic representations of programming languages can be proved formally. It is suggested that this is a suitable area for further research.

3.6-Discourse structure.

A mechanism for maintaining a structured interaction has been described. It is intended to be fairly general in nature, and to allow easy separation of domain-dependent and domain-independent knowledge involved in the interaction.

As it is currently implemented, the discourse mechanism is closely tied to the tutoring system, and lacks certain features which it needs for more general interactions (such as the ability to recognise pupil-initiated topic changes). Attention has not been paid to the details of generating individual utterances within this framework. It is suggested that the discourse mechanisms could usefully be explored in further detail, and that applying these ideas to the design of a stand-alone discourse package, which could discuss many domains, would be a fruitful reasearch topic.

This page
intentionally left blank.

This page
intentionally left blank

REFERENCES
==========

[Abelson 1980] A.diSessa,H.Abelson - Turtle Geometry.
MIT Press 1980

[Anderson 1982] J.R.Anderson - Acquisition of a cognitive skill.
Psychological Review Vol 89 1982

[Apt 1981] K.R.Apt - Ten years of Hoare's logic:a survey Part 1
ACM Transactions Vol.3 No.4 1981

[Archer 1916] R.L.Archer - Rousseau on education.
Arnold. 1916

[Ashlock 1976] R.B.Ashlock - Error patterns in computation.
Merrill 1976

[Bobrow 1977] D.G.Bobrow,R.M.Kaplan,M.Kay,D.A.Norman,H.Thompson, T.Winograd - GUS:A frame driven dialogue system.
Artificial Intelligence 8 1977

[Borger 1966] R.Borger,A.E.M.Seaborne - The psychology of learning.
Pelican 1966

[Bower 1981] G.H.Bower,E.R.Hilgard - Theories of learning (5th edition).
Prentice-Hall 1981

[Broudy 1969] H.Broudy - Socrates and the teaching machine.
in The Human Encounter Eds. S.Stoff and S.Schwartzberg
Harper-Row 1969

[Bruner 1966] J.Bruner - Towards a theory of instruction.
Harvard University Press 1966

[Burton 1982] R.R.Burton,J.S.Brown - An investigation of computer coaching for informal learning activities.
in Intelligent Tutoring Systems Eds. D.Sleeman and J.S.Brown
Academic Press 1982

[Carbonell 1983] J.G.Carbonell - Learning by analogy.
in Machine Learning. eds.R.S.Michalski,J.G.Carbonell,T.M.Mitchell.
Tioga 1983

[Carbonell 1970] J.R.Carbonell - Mixed initative man-computer instructional dialogues.
Bolt,Beranek and Newman Report.1971 1970

[Cerri 1983] S.A.Cerri,M.Fabrizzi,G.Marsali - The rather intelligent little lisper.
in Proceedings of AISB conference
on A.I. and Education 1983
1983

[Church 1956] A.Church - Introduction to mathematical logic.
Princeton University Press 1956

[Clancey 1979] W.J.Clancey - Dialogue management for rule-based tutorials.
in Proceedings of Sixth International Joint Conference on Artificial Intelligence 1979
ALSO Stanford working paper HPP-79-9

[Clint 1972] M.Clint,C.A.R.Hoare - Program proving:jumps and functions.

Acta Informatica 1 1972

[Cohen 1983] G.Cohen - The psychology of cognition.
Academic 1983

[Collins 1980] A.Collins,A.L.Stevens - Goals and strategies of interactive teachers.
Bolt,Beranek and Newman report 4345 1980

[Collins 1982] A.Collins,A.L.Stevens - A cognitive theory of interactive teaching.
in Instructional design theories and models. Ed. C.M.Reighluth

[Davey 1978] A.Davey - Discourse production.
Edinburgh University press 1978

[Delamont 1976] S.Delamont - Interaction in the classroom.
Metheun 1976

[Dietterich 1983] T.G.Dietterich,R.S.Michalski - A comparative review of selected methods of learning from examples.
in Machine Learning. eds.R.S.Michalski,J.G.Carbonell,T.M.Mitchell.
Tioga 1983

[Doyle 1973] J.F.Doyle (ed) - Educational Judgements.
Routledge & Kegan Paul 1973

[DuBoulay 1980] B.Du Boulay,T.O'Shea,J.Monk - The black box inside the glass box.
Edinburgh University DAI 133 1980

[Elsom-Cook 1982] M.T.Elsom-Cook - Matilda: A menu and template interface to Lisp.
M.Sc Thesis. Warwick University. 1982

[Elsom-Cook 1983] M.T.Elsom-Cook - A user interface for a Lisp teaching system.
Proceedings of the Ergonomics society conference: The User Interface.
September 1983

[Ennals 1981] J.R.Ennals - Logic as a computer language for children
Imperial College, Doc 81/6 1981

[Faught 1977] ?.Faught - Motivation and intensionality in a computer simulation model.
AIM - 305 Stanford University. 1977 [Feigenbaum 1963] E.A.Feigenbaum - The simulation of verbal learning behaviour.
in E.A.Feigenbaum,J.Feldman (eds)
Computers and thought.
McGraw-Hill 1963

[Floyd 1967] R.W.Floyd - Assigning meaning to programs.
Proceedings of symposia in applied maths 17 1967

[Furlong 1976] V.Furlong - Interaction sets in the classroom.
in M.Stubbs, S.Delamont (eds)
Explorations in classroom observation. 1976

[Goldstein 1980] I.P.Goldstein - Summary of Mycroft.
Artificial Intelligence 6 1980

[Goldstein 1982] I.P.Goldstein - The genetic graph: a representation for the evolution of procedural knowledge.
in Intelligent Tutoring Systems Eds. D.Sleeman and J.S.Brown

Academic Press 1982

[Gordon 1979] M.J.C.Gordon - The denotational description of programming languages.
Springer-Verlag 1979

[Grosz 1979] B.J.Grosz - The representation and use of focus in dialogue understanding.
Ph.D Dissertation 1979

[Hansen 1971] W.J.Hansen - Creation of hierarchic text with a computer display.
Ph.D thesis Stanford University 1971

[Hoare 1969] C.A.R.Hoare - An axiomatic basis for computer programming.
Communications of the ACM. Vol.12 No.10 1969

[Hoare 1972] C.A.R.Hoare - Proof of correctness of data representations.
Acta Informatica 1 1972

[Hoare 1973] C.A.R.Hoare,N.Wirth - An axiomatic definition of the programming language Pascal.
Acta Informatica 2 1973

[Hoare 1974] C.A.R.Hoare,P.E.Lauer - Consistent and complimentary formal theories of the semantics of programming languages.
Acta Informatica 3 1974

[Holt 1965] J.Holt - How children fail.
Pitman 1965

[Howe 1982] J.A.M.Howe,P.M.Ross,K.R.Johnson,F.Plane,R.Inglis - Programming in the classroom.
DAI report 157 Edinburgh University. 1982

[Huxley 1932] A.Huxley - Brave New World.
Chatto and Windus 1932

[Landin 1964] P.J.Landin - The mechanical evaluation of expressions.
Computer Journal Vol.6 1964

[Lenat 1983] D.B.Lenat - The role of heuristics in learning by discovery.
in Machine Learning. eds.R.S.Michalski,J.G.Carbonell,T.M.Mitchell.
Tioga 1983

[Levin 1980] J.A.Levin,J.A.Moore - Dialogue Games: metacommunication structures for natural language interaction.
Cognitive Science 11 1980

[Locke 1706] J.Locke - An essay concerning human understanding. (1706)
Dent Dutton 1976

[Mackenzie 1967] R.F.Mackenzie - The sins of the children.
Collins 1967

[Medina-Mora 1982] R.Medina-Mora - Syntax-directed editing: Towards integrated programming environments.
Dept. of Computer Science. Carnegie-Mellon 1982

[Michalski 1983] R.S.Michalski - A theory and methodology of inductive learning.
in Machine Learning. eds.R.S.Michalski,J.G.Carbonell,T.M.Mitchell.
Tioga 1983

[Neill 1968] A.S.Neill - Summerhill.
Pelican books 1968

[O'Connor 1956] D.J.O'Connor - An introduction to the philosophy of Education.
Routledge & Kegan Paul 1956

[O'Donnell 1982] M.J.O'Donnell - A critique of the foundations of Hoare style programming logics.
Communications of the ACM. 1982

[Orwell 1948] G.Orwell - 1984
Penguin 1948

[Papert 1971] S.Papert - Teaching children thinking.
MIT AIM-247 1971

[Papert 1980] S.Papert - Mindstorms.
Harvester Press 1980

[Peters 1966] R.S.Peters - Ethics and Education.
Allen & Unwin 1966

[Peters 1973]
in J.F.Doyle (ed) - Educational Judgements.
Routledge & Kegan Paul 1973

[Power 1979] R.Power - The organisation of purposeful dialogues.
Linguistics 17 1979

[Radcliffe Richards 1980] J.Radcliffe Richards - The sceptical feminist.
Routledge & Kegan Paul 1980

[Reichmann 1978] R.Reichmann - Conversational coherency.
Cognitive Science 2 1978

[Rich 1981] C.Rich - Inspection methods in programming.
MIT AI-TR-604 1981

[Robinson 1979] J.A.Robinson - Logic:form and function
Edinburgh University Press 1979

[Robinson 1982] J.Robinson - Diagram: A grammar for dialogues.
Communications of the ACM. Vol.25 No.1 1982

[Rousseau 1762] J.J.Rousseau - Emile. (1762)
Everyman edition. 1974

[Rusk 1967] R.R.Rusk - Doctrines of the great educators.
Macmillan. 1967

[Schegloff 1973] E.Schegloff,H.Sacks - Opening up closings.
Semiotica 8 1973

[Schneiderman 1980] B.Schneiderman - Software psychology.
Winthrop 1980

[Sime 1973] M.E.Sime,T.R.G.Green,D.J.Guest - Scope marking in computer conditionals: a psyvhological evaluation.
International Journal of Man-Machine studies. Vol.5 1973

[Simon 1983] H.A.Simon - Why should machines learn?
in Machine Learning. eds.R.S.Michalski,J.G.Carbonell,T.M.Mitchell.
Tioga 1983

[Soar 1972] R.S.Soar - Teacher behaviour related to pupil growth.
International review of education 18 1972

[Solomon 1976] C.Solomon,S.Papert - Case study of a child doing turtle graphics in LOGO.
MIT AIM-375 1976

[Stevens 1982] A.Stevens,A.Collins,S.E.Goldin - Misconceptions in students understanding.
in Intelligent Tutoring Systems
Eds. D.Sleeman and J.S.Brown Academic Press 1982

[Toffler 1970] A.Toffler - Future shock
Bodley Head 1970

[VanLehn 1981] K.VanLehn - Bugs are not enough: Empirical studies of bugs, impasses and repairs in procedural skills.
Xerox Cognitive and Instructional Sciences report 11
XEROX PARC 1981

[Weinberg 1971] G.M.Weinberg - The psychology of computer programming.
Van Nostrand Reinhold 1971

[Weiser 1982] M.Weiser - Programmers use slices when debugging.
Communications of the ACM. Vol.25 No.7 1982

[Whitehead 1932] A.N.Whitehead - The aims of education.
Benn 1950

[Winston 1981] P.H.Winston,B.K.Horn - Lisp.
Addison-Wesley 1981

[Zislis 1975] P.M.Zislis - Semantic decomposition of computer programs.
Acta Informatica 4 1975

APPENDICES

There are six appendices, as follows;

1) Summary of semantic primitives.

2) A semantic definition of LISP.

3) An interpreter for the semantics.

4) Output from a simple help package.

5) A problem solver.

6) A learning program.

Appendix 1 - Formal specification of semantic representation.

This appendix lists the basic units of the semantic representation in a concise form, with a brief description of each.

Overview.

Each statement in the language is represented by an "effect" predicate which corresponds to a semantic schema. For a particular instance of a statement the schema can be instantiated to provide a representation of the effect of that statement. The "effect" predicate provides an abstract syntax for the statement, a list of preconditions for the statement, and a body. The preconditions and body comprise a sequence of "primitive" operations, which may be regarded as component procedures or as declarative assertions about the state of the language processor. A semantically correct program is one in which all assertions can be shown to be true.

This basic representation may be reasoned about in a variety of ways. Some reasoning requires extra domain-specific knowledge. This is incorporated in the form of annotations associated with each primitive. These primitives and annotations are listed here.

Primitives.

** MISCELLANEOUS **

```
PRIMITIVE:TYPE(OBJECT,TYPE)
HELP-TEXT: " It checks that OBJECT is of type TYPE."
ERROR-TEXT: " OBJECT is not a TYPE"
INPUT:(1 1)
TYPE:(any tag)
REWRITE:()
```

This predicate must be defined as part of the target language specification. It defines each target language type in terms of the primitive datatypes of the semantics (identifier,number,ordered-set). It is true provided that "object" is of type "type".

```
PRIMITIVE:DO(EXPRESSION)
HELP-TEXT: " It evaluates EXPRESSION."
ERROR-TEXT: "  Cannot evaluate EXPRESSION"
INPUT:(1)
TYPE:(any)
REWRITE:()
```

This predicate determines the effect of its argument.

```
PRIMITIVE:OR(LIST1,LIST2)
HELP-TEXT: " At this point one of two things happens: EITHER (help-each LIST1)
OR, (help-each LIST2)."
ERROR-TEXT: " Cannot LIST1 or LIST2"
INPUT:(1 1)
TYPE:(o-set o-set)
REWRITE:()
```

```
PRIMITIVE:OR(LIST1,LIST2,LIST3)
HELP-TEXT: "At this point one of three things happens:
      1: (help-each ARG1)
      2: (help-each ARG2)
      3: (help-each ARG3)"
ERROR-TEXT: " Cannot LIST1 or LIST2 or LIST3"
INPUT:(1 1 1)
TYPE:(o-set o-set o-set)
REWRITE:()
```

Two versions of "or" which expects list of primitives for arguments. "or" only fails if all it's arguments fail.

```
PRIMITIVE:EQUALL(ARG1,ARG2)
HELP-TEXT: "A test is made to see if ARG1 is the same as ARG2."
ERROR-TEXT: "ARG1 is not the same as ARG2"
INPUT:(1 1)
TYPE:(any any)
REWRITE:()
```

Succeeds if both arguments are the same.

```
PRIMITIVE:DONOTHING
HELP-TEXT: " Nothing else is done."
ERROR-TEXT: ()
INPUT:()
TYPE:()
REWRITE:()
```

Does nothing.

** COMMUNICATION **

```
PRIMITIVE:IN(CHANNEL,VARIABLE)
HELP-TEXT: " It retrieves the ARG1 and calls it ARG2."
ERROR-TEXT: "  Cannot find a CHANNEL"
INPUT:(1 0)
TYPE:(tag id)
REWRITE:()
```

Get an item from the communication stream called "channel" and give it the name "variable".

```
PRIMITIVE:OUT(CHANNEL,ITEM)
HELP-TEXT: " It returns a ARG1 which is ARG2."
ERROR-TEXT: " Cannot put VARIABLE on CHANNEL"
INPUT:(1 1)
TYPE:(tag any)
REWRITE:()
```

Put "item" onto "channel".

** NUMERIC OPERATIONS **

All arithmetic functions provide integer operations only.

```
PRIMITIVE:GREATER(NUM1,NUM2)
HELP-TEXT: " It tests whether ARG1 is larger than ARG2."
```

```
ERROR-TEXT: " ARG1 is not greater than ARG2"
INPUT:(1 1)
TYPE:(number number)
REWRITE:()
```

Succeeds if num1>num2

```
PRIMITIVE:DIFFERENCE(NUM1,NUM2,RESULT)
HELP-TEXT: " NUM2 is subtracted from NUM1 to give RESULT."
ERROR-TEXT: " Cannot subtract NUM2 from NUM1"
INPUT:(1 1 0)
TYPE:(number number number)
REWRITE:()
```

result=num1-num2

```
PRIMITIVE:PLUS(NUM1,NUM2,RESULT)
HELP-TEXT: " NUM1 and NUM2 are added together to give RESULT"
ERROR-TEXT: " Cannot add NUM1 to NUM2"
INPUT:(1 1 0)
TYPE:(number number number)
REWRITE:()
```

result=num1+num2

```
PRIMITIVE:DIVIDE(TOP,BOTTOM,RESULT)
HELP-TEXT: " TOP is divided by BOTTOM to give RESULT."
ERROR-TEXT: " Cannot divide TOP by BOTTOM"
INPUT:(1 1 0)
TYPE:(number number number)
```

result=top/bottom

```
PRIMITIVE:TIMES(NUM1,NUM2,RESULT)
HELP-TEXT: " NUM1 is multiplied by NUM2 to give RESULT"
ERROR-TEXT: " Cannot multiply NUM1 by NUM2"
INPUT:(1 1 0)
TYPE:(number number number)
REWRITE:()
```

num1*num2=result

** SET OPERATIONS **

The ordered-set is the only structured datatype. Particular ordered-sets may have target-language types, but these do not affect the semantic processing.

```
PRIMITIVE:SETLENGTH(SET,NAME)
HELP-TEXT: " The length of ARG1 is determined. It is ARG2."
ERROR-TEXT: " Cannot determine the length of NAME"
INPUT:(1 0)
TYPE:(o-set number)
REWRITE:()
```

Given an ordered-set "set", this assigns "name" to be the length of that set.

PRIMITIVE:ELEMENT(N,SET,NAME)
HELP-TEXT: " The ARG1 element of ARG2 is found. It is ARG3"
ERROR-TEXT: " Cannot extract the N element of SET"
INPUT:(1 1 0)
TYPE:(number o-set any)
REWRITE:(element(N+X,S,M)==after(X-1,S,T),element(N,T,M))

Assigns the name NAME to the Nth element of SET.

PRIMITIVE:BEFORE(N,SET,NAME)
HELP-TEXT: " Those elements of ARG2 which come before the ARG1 element are found, and called ARG3."
ERROR-TEXT: " Cannot find the elements of SET which precede the Nth"
INPUT:(1 1 0)
TYPE:(number o-set any)
REWRITE:()

NAME becomes a set of those elements which precede the Nth element in SET.

PRIMITIVE:AFTER(N,SET,NAME)
HELP-TEXT: " Those elements of ARG2 which come after the ARG1 element are found, and called ARG3."
ERROR-TEXT: " Cannot find the elements of SET which follow the Nth"
INPUT:(1 1 0)
TYPE:(number o-set any)
REWRITE:()

NAME becomes a set of those elements which follow the Nth element in SET.

PRIMITIVE:ADDELEMENT(N,ITEM,SET,NAME)
HELP-TEXT: " ARG2 is added as the ARG1 element of ARG3. The result is called ARG4."
ERROR-TEXT: " Cannot add ITEM as the N element of SET"
INPUT:(1 1 1 0)
TYPE:(number any o-set o-set)
REWRITE:()

NAME is a new set, like SET but with an additional element ITEM in the Nth position.

PRIMITIVE:ADDBEFORE(N,NEWSET,SET,NAME)
HELP-TEXT: " The elements of ARG2 are added before the ARG1 element of ARG3. The result is called ARG4."
ERROR-TEXT: " Cannot add NEWSET before the N element of SET"
INPUT:(1 1 1 0)
TYPE:(number o-set o-set o-set)
REWRITE:(addbefore(X,A,B,C)==addafter(X-1,A,B,C))

NAME becomes a set, like SET but with the elements of NEWSET added before the Nth position.

PRIMITIVE:ADDAFTER(N,NEWSET,SET,NAME)
HELP-TEXT: " The elements of ARG2 are added after the ARG1 element of ARG3. The result is ARG4."
ERROR-TEXT: " Cannot add NEWSET after the N element of SET"
INPUT:(1 1 1 0)
TYPE:(number o-set o-set o-set)
REWRITE:(addafter(X,A,B,C)==addbefore(X+1,A,B,C))

Like ADDBEFORE, but the new elements appear after the Nth position.

** MEMORY OPERATIONS **

These primitives maintain representations of side-effects such as changing the value of a variable.

```
PRIMITIVE:CREATE(IDENTIFIER,TAGFIELD)
HELP-TEXT: " ARG1 is declared as a ARG2  variable."
 OR  " ARG1 is assigned a new value ARG2."
ERROR-TEXT: " Cannot declare ARG1 as an ARG2 variable"
  OR   " Cannot assign a value to ARG1"
INPUT:(1 1)
TYPE:(id any)
REWRITE:()
```

Produces a new item in environment with id and tagfield.

```
PRIMITIVE:DELETE(IDENTIFIER,PATTERN)
HELP-TEXT: " ARG1 is no longer a ARG2 variable."
 OR " The current value of ARG1 is forgotten."
ERROR-TEXT: " Cannot undeclare ARG1 as an ARG2 variable"
  OR   " Cannot remove the value of ARG1"
INPUT:(1 10)
TYPE:(id any)
REWRITE:()
```

Removes the first matching expression. Often needs wildcarding.

```
PRIMITIVE:SEE(IDENTIFIER,PATTERN)
HELP-TEXT: " ARG1 is tested to see if it is a ARG2 variable."
 OR "  The value of ARG1 is found to be ARG2."
ERROR-TEXT: " ARG1 is not an ARG2 variable"
 OR " CAnnot find the value of ARG1"
INPUT:(1 0)
TYPE:(id,any)
REWRITE:()
```

Attempts a prolog pattern match on items in environment.

```
PRIMITIVE:CHANGE(IDENTIFIER,PATTERN)
HELP-TEXT: " The value of ARG1 is changed to ARG2."
ERROR-TEXT:" Cannot assign a value to ARG1"
INPUT:(1 1)
TYPE:(id any)
REWRITE:(change(A,B)==delete(A,_),create(A,B))
```

Removes the first matching pattern and replaces it with this one (I haven't specified this properly!).

An Example definition.

Appendix 7 gives a (fairly) complete description of the semantics of Lisp. It uses some special identifiers (surrounded by $ signs) to represent particular constants. These identifiers are listed here;

****** Channels ******

"$value$"
"$input$"
"$output$"

These are names for channels of communication. $value$ is the channel along which expressions communicate, while $input$ and $output$ are side-effect channels used by READ and PRINT statements.

****** Types ******

"$sexpr$" "$number$" "$list$" "$atom$" "$boolean$"

These identifiers correspond to the lisp datatypes. They are defined in the "type" predicate.

****** Misc ******

"$expr$"

This is used to store user-defined functions. It also indicates that they are eval-spread.

"$prop$"

All property list additions are prefixed with this identifier.

"$global$"
"$fluid$"
"$local$"

These identifiers represent particular types of variable scoping. They are primarily used by assignment statements.

"$newline$"

This identifier can be placed on the output channel. It delimits lines of output.

Appendix 2 - Semantic description of Lisp.

```
/* This file contains axiomatic definitions of the semantics of */
/* lisp functions together with the language dependant support */
/* routines. */
/* Clean semantics version. */

/* ABS */

effect(abs(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARG1,'$number$'),
        or([greater(VARG1,0),out('$value$',VARG1)],
           [difference(0,VARG1,RES),out('$value$',RES)])]).

/* ADD1 */

effect(add1(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARG1,'$number$'),
        plus(1,VARG1,RES),
        out('$value$',RES)]).


/* AND */

/* APPEND */
effect(append(ARG1,ARG2),
       [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARG1,'$list$'),
        do(ARG2),
        in('$value$',VARG2),
        type(VARG2,'$list$'),
        addbefore(1,VARG1,VARG2,RES),
        out('$value$',RES)]).

/* APPLY */

/* ASSOC */

/* ATOM */

effect(atom(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
       in('$value$',VARG1),
       or([type(VARG1,'$atom$'),out('$value$',t)],
          [out('$value$',nil)])]).

/* CAR */

effect(car(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
       in('$value$',VARG1),
       type(VARG1,'$list$'),
       element(1,VARG1,VAL),
       out('$value$',VAL)]).
```

```
/* CDR */
effect(cdr(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
       in('$value$',VARG1),
       type(VARG1,'$list$'),     /* what about empty lists? */
       after(1,VARG1,VAL),
       out('$value$',VAL)]).

/* CONS */
effect(cons(ARG1,ARG2),
       [type(ARG1,'$sexpr$'),
        type(ARG2,'$sexpr$')],
       [do(ARG1),
       in('$value$',VARG1),
       type(VARG1,'$sexpr$'),
       do(ARG2),
       in('$value$',VARG2),
       type(VARG2,'$list$'),     /* what about empty lists? */
       addelement(1,VARG1,VARG2,RES),
       out('$value$',VAL)]).



/* COND  - simple version, fixed arity; true booleans */

effect(cond(ARG1,ARG2,ARG3,ARG4),
       [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$'),
        type(ARG3,'$sexpr$'),type(ARG4,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        or([equall(VARG1,t),do(ARG2),    /* not right! */
            in('$value$',VARG2),out('$value$',VARG2)],
           [do(ARG3),
            in('$value$',VARG3),
            or([equall(VARG3,t),do(ARG4),    /* not right! */
                in('$value$',VARG4),out('$value$',VARG4)],
               [out('$value$',nil)])])]).

/* DE */
effect(de(ARG1,ARG2,ARG3),
        [type(ARG1,'$atom$'),type(ARG2,'$list$'),type(ARG3,'$sexpr$')],
        [create(ARG1,['$expr$',ARG2,ARG3]),
         out('$value$',ARG1)]).

/* DIFFERENCE */

effect(difference(ARG1,ARG2),
       [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARG1,'$number$'),
        do(ARG2),
        in('$value$',VARG2),
        type(VARG2,'$number$'),
        difference(VARG1,VARG2,RES),
        out('$value$',RES)]).


/* EQUAL  */

effect(equal(ARG1,ARG2),
       [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$')],
       [do(ARG1),
```

```
        in('$value$',VARG1),
        type(VARG1,'$sexpr$'),
        do(ARG2),
        in('$value$',VARG2),
        type(VARG2,'$sexpr$'),
        or([equall(VARG1,VARG2),out('$value$',t)],
           [out('$value$',nil)])]).

/* EVAL */

effect(eval(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARg1,'$sexpr$'),
        do(VARG1),
        in('$value$',VVARG1),
        out('$value$',VVARG1)]).


/* GET */   /* Arg1 is variable, arg2 is indicator. */
effect(get(ARG1,ARG2),  /* is records & properties together a problem? */
       [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARG1,'$atom$'),
        do(ARG2),
        in('$value$',VARG2),
        type(VARG2,'$atom$'),
        see(VARG1,['$prop$',VARG2,RES]),
        out('$value$',RES)]).



/* GLOBAL  - wrong arity */ /* Only declare if not already there. */
effect(global(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARG1,'$atom$'),
        create(VARG1,'$global$')]).


/* GLOBALP */

effect(globalp(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        or([see(VARG1,'$global$'),out('$value$',t)],
           [out('$value$',nil)])]).

/* GREATERP */
effect(greaterp(ARG1,ARG2),
       [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$')],
       [do(ARG1),
         in('$value$',VARG1),
         type(VARG1,'$number$'),
         do(ARG2),
         in('$value$',VARG2),
         type(VARG2,'$number$'),
         or([greater(VARG1,VARG2),out('$value$',t)],
            [out('$value$',nil)])]).
```

```
/* LENGTH */
effect(length(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARG1,'$list$'),
        setlength(VARG1,RES),
        out('$value$',RES)]).

/* LIST - wrong arity */
effect(list(ARG1,ARG2),
        [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$')],
        [do(ARG1),
         in('$value$',VARG1),
         type(VARG1,'$list$'),
         do(ARG2),
         in('$value$',VARG2),
         type(VARG2,'$list$'),
         addelement(1,VARG1,NULSET,LST), /* whoops */
         addelement(2,VARG2,LST,RES),
         out('$value$',RES)]).


/* MAPCAR */

/* MAPLIST */

/* MAX2 */

effect(max2(ARG1,ARG2),
       [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$')],
       [do(ARG1),
         in('$value$',VARG1),
         type(VARG1,'$number$'),
         do(ARG2),
         in('$value$',VARG2),
         type(VARG2,'$number$'),
         or([greater(VARG1,VARG2),out('$value$',VARG1)],
            [out('$value$',VARG2)])]).


/* MIN2 */

effect(min2(ARG1,ARG2),
       [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$')],
       [do(ARG1),
         in('$value$',VARG1),
         type(VARG1,'$number$'),
         do(ARG2),
         in('$value$',VARG2),
         type(VARG2,'$number$'),
         or([greater(VARG2,VARG1),out('$value$',VARG1)],
            [out('$value$',VARG2)])]).


/* NOT  */

effect(not(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        or([equall(VARg1,nil),out('$value$',t)],
           [out('$value$',nil)])]).
```

```
/* NULL */
effect(null(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        or([equall(VARg1,nil),out('$value$',t)],
           [out('$value$',nil)])]).

/* NUMBERP */

effect(numberp(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
       in('$value$',VARG1),
       or([type(VARG1,'$number$'),out('$value$',t)],
          [out('$value$',nil)])]).


/* OR */

/* PLUS2 */
effect(plus2(ARG1,ARG2),
       [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARG1,'$number$'),
        do(ARG2),
        in('$value$',VARG2),
        type(VARG2,'$number$'),
        plus(VARG1,VARG2,RES),
        out('$value$',RES)]).

/* PRINT */

effect(print(ARG1),
       [type(ARG1,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARG1,'$sexpr$'),
        out('$output$',VARG1),
        out('$output$','$newline$'),
        out('$value$',VARG1)]).

/* PROG */
/* Test case! One variable declaration and one bit of body! */
effect(prog(ARG1,ARG2),
       [type(ARG1,'$atom$'),
        type(ARG2,'$sexpr$')],
       [create(ARG1,'$local$'),
        create(ARG1,['$value$',nil]),
        do(ARG2),
        in('$value$',VARG2),
        delete(ARG1,['$value$',_]),
        delete(ARG1,'$local$'),
        out('$value$',VARG2)]).


/* PROGN */

/* PUT */  /* problem -what if it already exists? (it does!) */
effect(put(ARG1,ARG2,ARG3), /* 1 is atom, 2-ind,3-data */
       [type(ARG1,'$sexpr$'),
        type(ARG1,'$sexpr$'),
```

```
        type(ARG1,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARG1,'$atom$'),
        do(ARG2),
        in('$value$',VARG2),
        type(VARG2,'$atom$'),
        do(ARG3),
        in('$value$',VARG3),
        type(VARG3,'$sexpr$'),
        change(VARG1,['$prop$',VARG2,VARG3])]).


/* QUOTE */

effect(quote(ARG1),
       [type(ARG1,'$sexpr$')],
       [out('$value$',ARG1)]).


/* QUOTIENT */
effect(quotient(ARG1,ARG2),
       [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARG1,'$number$'),
        do(ARG2),
        in('$value$',VARG2),
        type(VARG2,'$number$'),
        divide(VARG1,VARG2,RES),
        out('$value$',RES)]).



/* READ */
effect(read,
        [],
        [in('$input$',VAL),
         out('$value$',VAL)]).


/* REVERSE */

/* SET */
effect(set(ARG1,ARG2),
       [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$')],
       [do(ARG1),
        in('$value$',VARG1),
        type(VARG1,'$atom$'),
        or([see(ARG1,'$global$')],
           [see(ARG2,'$fluid$')],
           [see(ARG3,'$local$')]),
        do(ARG2),
        in('$value$',VARG2),
        change(VARG1,['$value$',VARG2]),
        out('$value$',VARG2)]).


/* SETQ */

effect(setq(ARG1,ARG2),
       [type(ARG1,'$atom$'),
        type(ARG2,'$sexpr$'),
        or([see(ARG1,'$global$')],
```

```
            [see(ARG2,'$fluid$')],
            [see(ARG3,'$local$')])],
        [do(ARG2),
         in('$value$',VARG2),
         type(VARG2,'$sexpr$'),
         change(ARG1,['$value$',VARG2]),
         out('$value$',VARG2)]).

/* SUB1 */

effect(sub1(ARG1),
        [type(ARG1,'$sexpr$')],
        [do(ARG1),
         in('$value$',VARG1),
         type(VARG1,'$number$'),
         difference(VARG1,1,RES),
         out('$value$',RES)]).

/* TIMES */
effect(times(ARG1,ARG2),
        [type(ARG1,'$sexpr$'),type(ARG2,'$sexpr$')],
        [do(ARG1),
         in('$value$',VARG1),
         type(VARG1,'$number$'),
         do(ARG2),
         in('$value$',VARG2),
         type(VARG2,'$number$'),
         times(VARG1,VARG2,RES),
         out('$value$',RES)]).



/* ****** More general matches  ********** */

/* Numbers and noeval atoms */

effect(NUM,
       [],
       [out('$value$',NUM)])
       :-('&type'(_,_,NUM,'$number$');'&type'(_,_,NUM,'$boolean$')).


/* Atoms  */

effect(ATOM,
        [or([see(ARG1,'$global$')],
            [see(ARG2,'$fluid$')],
            [see(ARG3,'$local$')])],
        [see(ATOM,['$value$',VAL]),
         out('$value$',VAL)])
      :- '&type'(_,_,ATOM,'$atom$'),not('&type'(_,_,ATOM,'$boolean$')).

/* ******  Object types   ************* */
/* These look funny for the interpreter. */
 '&type'(ENV,ENV,X,'$number$'):-integer(X).
 '&type'(ENV,ENV,t,'$boolean$').
 '&type'(ENV,ENV,nil,'$boolean$').
 '&type'(ENV,ENV,X,'$alpha$'):-atomic(X).

 '&type'(ENV,ENV,X,'$atom$'):-'&type'(ENV,ENV,X,'$number$');
                        '&type'(ENV,ENV,nil,'$boolean$');
                        '&type'(ENV,ENV,X,'$alpha$').
```

```
'&type'(ENV,ENV,COMPOUND,X):-functor(COMPOUND,'**list',_),
                             X='$list$'.
'&type'(ENV,ENV,X,'$sexpr$').
```

Appendix 3 - Interpreter for semantic definition.

```
/* A new pretty interpreter for the semantics. */
/* This must have a file of primitives and the semantics */
/* vis; ['fullisp.cp',shot1,prim1]     */

/* Set up default actions */
/* tracing on. */
bugged.
/* debug things */
/* debugger. */
/* report failures - if this isn't on it just continues -turned off inside or */
report.

/* Simple invocation for problems without environment */
/* e.g. do(quote(3),K).  */
do(X,M):-do_in_env([],M,X).


do_in_env(PREENV,POSTENV,STMT):- effect(STMT,PRETEST,POSTTEST),
                                 execute_each(PREENV,PRETEST,EENV),
                                 execute_each(EENV,POSTTEST,POSTENV),
                                 ((bugged,
                                   display_full(effect(STMT,PRETEST,POSTTEST),
                                                POSTENV));
                                   true).


/* This one does each argument in turn.  If report then */
/* it prints error information via display_report.   */
/* invoke like execute_each([],[do(3),in(value,VAL)],M). */
execute_each(ENV,[],ENV).
execute_each(ENV,[H|T],PENV):-H=..[FUNC|ARGS],
                              secretname(FUNC,SFUNC),
                              SEC=..[SFUNC,ENV,EENV|ARGS],
                              careful_call(SEC),
                              execute_each(EENV,T,PENV).

/* This gets a single predicate and executes it like careful_call(do(3)) */
careful_call(X):-call(X).
careful_call(X):-report,display_report(X),
                 ((debugger,debugcall(X));true).
careful_call(X):-not(report),true. /* continue on error! */


debugcall(X):-X=..[FUNC|ARGS],
              debugname(FUNC,DFUNC),
              DCALL=..[DFUNC|ARGS],
              call(DCALL).

/* ************  Boring low level bits ****************** */

/* A secret name is & on the front of the old name. */
secretname(X,Y):-name(X,XLIST),
                 name('&',[AND]),
                 name(Y,[AND|XLIST]).
/* A debugname is debug on the old name. */
debugname(X,Y):-name(X,XLIST),name('debug',DLIST),
                append(DLIST,XLIST,YLIST),name(Y,YLIST).


/* **************************  DISPLAY ROUTINES  *********** */

display_full(A,B):-write('STATEMENT: '),write(A),nl,
                   write('ENVIRONMENT: '),write(B),nl,nl.
```

```
/* The test has failed if it gets here. */
display_report(X):-write('cannot satisfy; '),nl,
                   write(X),nl.

/* ************************************* */
/* **** Some examples of the system running */

 | ?- do(quote(mary),M).
 STATEMENT: effect(quote(mary),
                  [type(mary,$sexpr$)],[out($value$,mary)])
 ENVIRONMENT: [[$value$,mary]]


 M = [[$value$,mary]]

 yes
 | ?- do(car(quote('**list'(jellyfish,eat,toast))),M).
 STATEMENT: effect(quote(**list(jellyfish,eat,toast)),
                  [type(**list(jellyfish,eat,toast),$sexpr$)],
                  [out($value$,**list(jellyfish,eat,toast))])
 ENVIRONMENT: [[$value$,**list(jellyfish,eat,toast)]]

 STATEMENT: effect(car(quote(**list(jellyfish,eat,toast))),
                  [type(quote(**list(jellyfish,eat,toast)),$sexpr$)],
                  [do(quote(**list(jellyfish,eat,toast))),
                   in($value$,**list(jellyfish,eat,toast)),
                   type(**list(jellyfish,eat,toast),$list$),
                   element(1,**list(jellyfish,eat,toast),jellyfish),
                   out($value$,jellyfish)])
 ENVIRONMENT: [[$value$,jellyfish]]


 M = [[$value$,jellyfish]]

 yes
 | ?- do(car(quote(mary)),M).
 STATEMENT: effect(quote(mary),
           [type(mary,$sexpr$)],[out($value$,mary)])
 ENVIRONMENT: [[$value$,mary]]

 cannot satisfy;
 &type([],_258,mary,$list$)
 cannot satisfy;
 &element(_258,_303,1,mary,_17)
 STATEMENT: effect(car(quote(mary)),
           [type(quote(mary),$sexpr$)],
           [do(quote(mary)),
            in($value$,mary),
            type(mary,$list$),
            element(1,mary,_17),
            out($value$,_17)])
 ENVIRONMENT: [[$value$,_17]|_303]


 M = [[$value$,_17]|_303]

 yes
```

```
4 lisp> (=> quote help)
 It checks that ARG1 is of type $SEXPR$.
 It returns a $VALUE$ which is ARG1.
NIL

5 lisp> (=> car help)
 It checks that ARG1 is of type $SEXPR$.
 It evaluates ARG1.
 It retrieves the $VALUE$ and calls it VARG1.
 It checks that VARG1 is of type $LIST$.
The 1 element of VARG1 is found.  It is VAL
 It returns a $VALUE$ which is VAL.
NIL

6 lisp> (=> cons help)
 It checks that ARG1 is of type $SEXPR$.
 It checks that ARG2 is of type $SEXPR$.
 It evaluates ARG1.
 It retrieves the $VALUE$ and calls it VARG1.
 It checks that VARG1 is of type $SEXPR$.
 It evaluates ARG2.
 It retrieves the $VALUE$ and calls it VARG2.
 It checks that VARG2 is of type $LIST$.
VARG1 is added as the 1 element of VARG2.  The result is called RES.
 It returns a $VALUE$ which is VAL.
NIL

7 lisp> (=> cond help)
 It checks that ARG1 is of type $SEXPR$.
 It checks that ARG2 is of type $SEXPR$.
 It checks that ARG3 is of type $SEXPR$.
 It checks that ARG4 is of type $SEXPR$.
 It evaluates ARG1.
 It retrieves the $VALUE$ and calls it VARG1.
At this point one of two things happens: EITHER A test is made to see if VARG1
 is the same as T
 It evaluates ARG2.
 It retrieves the $VALUE$ and calls it VARG2.
 It returns a $VALUE$ which is VARG2.
 OR, It evaluates ARG3.
 It retrieves the $VALUE$ and calls it VARG3.
At this point one of two things happens: EITHER
A test is made to see if VARG3
 is the same as T
 It evaluates ARG4.
 It retrieves the $VALUE$ and calls it VARG4.
 It returns a $VALUE$ which is VARG4.
 OR, It returns a $VALUE$ which is NIL.


NIL
% COND is not exactly the clearest description in the world!

8 lisp> (=> de help)
 It checks that ARG1 is of type $ATOM$.
 It checks that ARG2 is of type $LIST$.
 It checks that ARG3 is of type $SEXPR$.
ARG1 is assigned a new value ($EXPR$ ARG2 ARG3)
 It returns a $VALUE$ which is ARG1.
NIL
% DE is not very good.  It is too close to being
% implementation details.
```

```
9 lisp> (=> prog help)
 It checks that ARG1 is of type $ATOM$.
 It checks that ARG2 is of type $SEXPR$.
ARG1 is declared as a $LOCAL$ variable.
ARG1 is assigned a new value ($VALUE$ NIL)
 It evaluates ARG2.
 It retrieves the $VALUE$ and calls it VARG2.
 The current value of ARG1 is forgotten.
ARG1 is no longer a $LOCAL$ variable.
 It returns a $VALUE$ which is VARG2.
NIL

% This is the same package applied to the
% real expression (ATOM (CAR X)), where X=(FRED IS DEAD).
% It doesn't look at the works of CAR, but it could.
11 lisp> (=> expression help)
 It checks that (CAR X) is of type SEXPR.
 It evaluates (CAR X).
 It retrieves the VALUE and calls it FRED.
At this point one of two things happens: EITHER
It checks that FRED is of type
ATOM.
 It returns a VALUE which is T.
 OR,  It returns a VALUE which is NIL.

NIL
```

Appendix 5 - Lisp problem solver.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PROBLEM - SOLVER AS DESCRIBED IN SECTION 3.2     %
%    IMPLEMENTED BY IAN CARR.                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

(global '(!#rewrite))

% EXAMPLE OF REWRITE RULES.
(setq !#rewrite '( ((element #x #a #b)
                      ((after #X-n #a #z)(element #n #z #b)) )) )

% e.g. (element 2 (a b c) b) == ((after 1 (a b c) (b c))
%                                 (element 1 (b c) b))

(GLOBAL '(!#STACK))


(DE PROBLEM_SOLVER (PROBLEM)
    (PROGN (SETQ !#STACK NIL)
           (PROBLEM_LIMITER PROBLEM) ))


(DE PROBLEM_LIMITER (PROBLEM)
    (COND ((NULL PROBLEM) (PROGN (PRINT !#STACK) NIL))
          (T (SEARCH_FN_PRIMS (LIST_FN_PRIMS (CAR PROBLEM) !#LISP)
                              PROBLEM)) ))



(DE SEARCH_FN_PRIMS  (FN_PRIMS PROBLEM)
    (COND ((NULL FN_PRIMS) NIL)
          (T (PROGN (SOLVE (CAR FN_PRIMS) PROBLEM)
                    (SEARCH_FN_PRIMS (CDR FN_PRIMS) PROBLEM) )) ))

(DE SOLVE (FN_PRIM PROBLEM)
    (COND ((MATCH_PRIMS FN_PRIM PROBLEM)
           T )
          ((CAN_REWRITE (CAR PROBLEM) !#REWRITE)    %ONE REWRITE ONLY
           (PROBLEM_LIMITER (APPEND (CAN_REWRITE (CAR PROBLEM) !#REWRITE)
                                    (CDR PROBLEM)) ))
          (T NIL) ))



(DE MATCH_PRIMS (FN_PRIM PROBLEM)
    (PROGN
        (PUSH (CAR FN_PRIM))        % CAR
        (PUSH (CAR (CAR PROBLEM))) % ELEMENT
        (COND ((MATCH_EACH (CDR(CAR(CDR FN_PRIM)))
                           (CDR (CAR PROBLEM)))    % ( A B #C)
                T ))
        (POP)
        (POP)))

(DE MATCH_EACH (PRIM_ARGS PROB_ARGS)
    (COND ((NULL PRIM_ARGS) (PROBLEM_LIMITER (CDR PROBLEM)))
          ((MATCH_BIT_ANS (CAR PRIM_ARGS)
                          (CAR PROB_ARGS))
             (PROGN
                 (PUSH (LIST (CAR PRIM_ARGS)
                             (CAR PROB_ARGS)))
                 (COND ((MATCH_EACH (CDR PRIM_ARGS)
                                    (CDR PROB_ARGS))
```

```
                                    T)
                              (T (PROGN (POP) NIL)))))
          (T
             NIL)))


(DE POP NIL
   (SETQ !#STACK (CDR !#STACK)))

(DE PUSH (X)
   (SETQ !#STACK (CONS X !#STACK)))




(DE CAN_REWRITE (PRIM REWRITE_RULES)
   (COND ((NULL REWRITE_RULES) NIL)
         ((MATCH_REWRITE PRIM (CAR REWRITE_RULES)) (CAR REWRITE_RULES))
         (T (CAN_REWRITE PRIM (CDR REWRITE_RULES))) ))

(DE MATCH_REWRITE (PRIM RULE)
   (COND ((LIST_ANSWER PRIM (CAR RULE)) T)
         (T NIL) ))


%this function takes two primitives. if the second maps as a solution
%of the first, then it returns a list of the first, otherwise nil.
%NB. below her doesnt take elements containing embedding, ie OR... yet.
(DE LIST_ANSWER (PRIM SOLN?)
   (COND ((NULL (EQUAL (LENGTH PRIM)
                        (LENGTH SOLN?))) NIL)
         ((MATCH_ANSWER PRIM SOLN?) (LIST PRIM))
         (T NIL) ))

%this function takes two primitives. if the second maps as a solution
%to the first it returns T, otherwise nil.
%from here on down errors enter if not picked up by the equal elements
%test above, eg extra soln? elements undetected, and the checkvars fire t
%on locating a variable - they dont check if its present or not.
(DE MATCH_ANSWER (PRIM SOLN?)
   (COND ((NULL PRIM) T)
         ((NULL (MATCH_BIT_ANS (CAR PRIM) (CAR SOLN?))) NIL)
         (T (MATCH_ANSWER (CDR PRIM) (CDR SOLN?))) ))

%this function takes elements of two primitives. if the second is a
%possible instantiation of the first then t, otherwise nil.
(DE MATCH_BIT_ANS (BIT_PRIM BIT_SOLN?)
   (COND ((AND (CHECK_VAR BIT_PRIM) (CHECK_VAR BIT_SOLN?)) T)
         ((AND (NULL (CHECK_VAR BIT_PRIM)) (CHECK_VAR BIT_SOLN?)) T)
         ((EQUAL (NULL (CHECK_VAR BIT_PRIM)) (NULL(CHECK_VAR BIT_SOLN?))) T)
         (T NIL) ))

(DE MATCH_BIT_ANS (BIT_PRIM BIT_SOLN?)
   (COND ((CHECK_VAR BIT_PRIM) T)
         ((CHECK_VAR BIT_SOLN?) T)  %want to instantiate bit_soln?
                                    %as bit_prim
         ((EQUAL BIT_PRIM BIT_SOLN?) T)
         (T NIL) ))

%this function takes an s-expression. if it is given a variable (start with #)
%it returns t. if a constant, nil. if a list, it recursively hands it to CANT
(DE CHECK_VAR (BIT)
   (COND ((EQUAL (CAR (EXPLODE BIT)) '#) T)
```

```
        (T NIL) ))


%this function takes a primitive, and a list of
%function  definitions. It returns a list of those function
% names whose definitions contain one (or  more) of the given
% primitives, together with their located primitive.

(DE LIST_FN_PRIMS (PRIMITIVE FUNCTIONS)
    (COND ((NULL (CAR FUNCTIONS)) NIL)
          (T (APPEND (MATCH_FN PRIMITIVE (CAR FUNCTIONS))
                     (LIST_FN_PRIMS PRIMITIVE (CDR FUNCTIONS)) )) ))


%This function takes a primitive, and a function definition.
%It returns a list of lists of the function name and any primitive
% with the same name as the given primitive contained
% in its function  body, and nil otherwise.

(DE MATCH_FN (PRIMITIVE FUNCTION)
    (MAPCAR (MATCH_BODS (CAR PRIMITIVE) (CDR FUNCTION))
          '(LAMBDA (X) (LIST (CAR FUNCTION) X)) ))


%this function takes a primitive name, and a list of bodies and recurses
%along them checking the elements of the bodies for matches.

(DE MATCH_BODS (PRIMITIVE BODIES)
    (COND ((NULL BODIES) NIL) %    atom may be safer if bad list?
          ((MATCH_PRIM PRIMITIVE (CAR BODIES))
           (APPEND (MATCH_PRIM PRIMITIVE (CAR BODIES))
                   (MATCH_BODS PRIMITIVE (CDR BODIES)) ))
          (T (MATCH_BODS PRIMITIVE (CDR BODIES)) )) )


%this function takes a primitive name, and the body of a function definition.
%It returns the found primitive if a match with the given primitive name
%can be found in the definition, otherwise NIL.


(DE MATCH_PRIM (PRIMITIVE FN_BODY)       %good start
    (COND ((NULL FN_BODY) NIL)              %NULL stopping condition
          ((EQUAL PRIMITIVE (CAAR FN_BODY))
           (APPEND  (LIST (CAR FN_BODY))
                    (MATCH_PRIM PRIMITIVE (CDR FN_BODY)) )) %detect match
          ((EQUAL 'OR (CAAR FN_BODY))
           (APPEND (MATCH_BODS PRIMITIVE (CDAR FN_BODY))
                   (MATCH_PRIM PRIMITIVE (CDR FN_BODY)) ))
          (T (MATCH_PRIM PRIMITIVE (CDR FN_BODY)) ) ))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  SOME EXAMPLES

% This corresponds to requesting an expression which
% would return the first element of a list.
% The solution identifies the proposed function(s)
% in the solution, identifies the relevant primitive
% in that function, and presents a correspondence
% between items in the problem and items within the
% function.  The solution should be read right to left.

>> (PROBLEM_SOLVER '((ELEMENT 1 M #E ) ))
```

```
((#VAL #E) (#VARG1 M) (1 1) ELEMENT CAR)
NIL

% This is a slightly more complex problem.  There are two
% components to the problem; extracting an element and
% returning everything following the first item of that
%  element.  This solution
% corresponds to (CDR (CAR X))

>>  (PROBLEM_SOLVER '((ELEMENT 1 #F #O) (AFTER 1 #F O)))

((#VAL O) (#VARG1 #F) (1 1) AFTER CDR
 (#VAL #O) (#VARG1 #F) (1 1) ELEMENT CAR)
NIL

% In this problem the system offers two methods of adding
% something to the front of a list - CONS and LIST.

>>  (PROBLEM_SOLVER '((ADDELEMENT 1 #M #E #C) ))

((#RES #C) (#VARG2 #E) (#VARG1 #M) (1 1) ADDELEMENT CONS)
((#LST #C) (NULSET #E) (#VARG1 #M) (1 1) ADDELEMENT LIST)
NIL
```

Appendix 6 - A simple learning program.

```
/* given an expression and a resulting environment, hypothesise */
/* a semantic description.  ONE SHOT */

describe(EXPRESSION,RESULT,DESCRIPTION):- perceive(EXPRESSION,RESULT,
                                              effect(XP,PP)),
                                          generalize([XP|PP],DESC),
                                          DESCRIPTION=..[effect|DESC].


/* perceiving involves inserting OUT for any channel used */
/* type for any subexpression, and changememory for any side-effect. */

perceive(EXP,RES,effect(EXP,PEXP)):- side_effects(RES,SE_EXP),
                         outputs(RES,OUT_EXP),
                         append(SE_EXP,OUT_EXP,SO),
                         typing(EXP,EEXP),
                         append(EEXP,SO,PEXP).


/* add a change statement for each environment point other than channels */
side_effects([],[]).
side_effects([['$value$',_]|T],K):- side_effects(T,K). /* no channels */
side_effects([[A,B]|T],[change(A,B)|K]):- side_effects(T,K).
side_effects([_|T],K):- side_effects(T,K).

/* Add an OUT statement for each channel */
outputs([],[]).
outputs([['$value$',B]|T],[out('$value$',B)|K]):- outputs(T,K).
     /* no channels */
outputs([_|T],K):-outputs(T,K).

/* type arguments only */
typing(EXP,TYPES):-EXP=..[STMNT|ARGS],
                   type_each(ARGS,TYPES).

type_each([],[]).
type_each([H|T],[type(H,X)|Z]):-type(H,X),type_each(T,Z).


/* generalize by applying a rule to what you've got. */

generalize(EXP,DESC):-gen_rule(NAME,EXP,DESC).


/* Actual rules */

/* pick out constants which are repeated, replace with variables. */
/* RLIST looks like [[x,_23],[y,_25]] etc.      */
gen_rule(const_to_variable,EXP,RESULT):- find_repeats(EXP,RLIST),
                                         con_to_var(EXP,RLIST,RESULT).

/* find repeated elements by listing all constants and extracting doubles */
find_repeats(X,Y):-list_constants(X,CLIST),
                   flush(CLIST,CCLIST), /* remove system tags */
                   doubles(CCLIST,DLIST),
                   var_pair(DLIST,Y).


/* append args to each predicate. */
list_constants([],[]).
list_constants([X|Y],M):-X=..[_|ARGS],
                         list_constants(Y,YM),
                         append(ARGS,YM,M).
```

```
/* flush eliminates all terms beginning $ */
/* fix bug for non-atomic items (specially numbers) */

flush([],[]).
flush([H|T],Y):-name(H,[36|_]),flush(T,Y).   /* 36=$ */
flush([H|T],[H|Y]):-flush(T,Y).

/* produce a list of duplicates only. */
doubles([],[]).
                           /* no duplicates */
doubles([H|T],[H|M]):-member(H,T),
                      doubles(T,M),
                      not(member(H,M)).
doubles([_|T],M):-doubles(T,M).

/* is this the way to generate variable refs? */
var_pair([],[]).
var_pair([X|T],[[X,XVAR]|M]):-var_pair(T,M).

/* pick off each predicate and c_to_v on it */
con_to_var([],_,[]).
con_to_var([X|P],RLIST,[FIR|RES]):-X=..[FUNC|ARGS],
                                   c_to_v(ARGS,RLIST,RARG),
                                   FIR=..[FUNC|RARG],
                                   con_to_var(P,RLIST,RES).

/* single set of args converted to variables. */
c_to_v([],_,[]).
c_to_v([X|T],RLIST,[B|Y]):-member([X,B],RLIST),
                           c_to_v(T,RLIST,Y).
c_to_v([X|T],RLIST,[X|Y]):-c_to_v(T,RLIST,Y).


/* Trivia */

member(_,[]):-!,fail.
member(X,[X|_]).
member(X,[_|T]):-member(X,T).

append([],Z,Z).
append([X|Y],Z,[X|M]):-append(Y,Z,M).

type(t,'$boolean$').
type(nil,'$boolean$').
type(X,'$number$'):-nonvar(X),integer(X).
type(X,'$atom$'):-nonvar(X),atomic(X).
type(COMPOUND,X):-nonvar(COMPOUND),functor(COMPOUND,FUNC,_),
                    name('*',[STAR]),
                    name('$',[DOLLAR]),
                    name(FUNC,[STAR,STAR|FL]),
                    name(X,[DOLLAR|FL]).
type(X,'$sexpr$').

/* ******** BEYOND THIS POINT ARE EXAMPLES ****** */

| ?- perceive(quote(9),[['$value$',9]],M).

M = effect(quote(9),[[type(9,atom)],[],out($value$,9)]])

yes

| ?- describe(quote(5),[['$value$',5]],K).
```

```
K = effect(quote(_179),type(_179,$atom$),out($value$,_179))

yes

| ?- perceive(setq(x,3),[[x,['$value$',3]],['$value$',3]],M).

M = effect(setq(x,3),
          [[type(x,atom),type(3,atom)],
          [change(x,[$value$,3])],out($value$,3)])

yes

| ?- describe(setq(x,5),[[x,['$value$',5]],['$value$',5]],K).

K = effect(setq(_330,_336),
          type(_330,$atom$),
          type(_336,$atom$),
          change(_330,_336),
          out($value$,_336))

yes

/* To show there is nothing up my sleeve, here */
/* is something called CONS which is nothing like LISP */

| ?- perceive(cons(t,6),
              [[x,'**list'(t,3)],['$value$','**list'(t,66)]],M).

M = effect(cons(t,6),
          [[type(t,boolean),type(6,atom)],
          [change(x,**list(t,3))],out($value$,**list(t,66))])

yes
```