LEARNING AND USING PROLOG:
An Empirical Investigation

Josie Taylor and Benedict du Boulay

1987

LEARNING AND USING PROLOG:
An Empirical Investigation

Josie Taylor and Benedict du Boulay

School of Cognitive Sciences
University of Sussex
Falmer Brighton
August 1987

1.0 INTRODUCTION

This report outlines the conclusions of SERC research project (GR/D/20328) to investigate the difficulties experienced by beginners and experts learning and using the logic-based programming language PROLOG,

The original proposal outlined a strategy for investigating different kinds of users - i.e. Prolog experts, experts in languages other than Prolog, and complete novices. It was hoped at that time that we could also sample populations of users from industry, instead of confining the investigation to academic users. In the event, the main thrust of the research has concentrated on students who were novice Prolog programmers. Much of the theoretical and empirical work on these novices have been reported in Taylor (1987), where a framework for identifying various categories of errors is described, and in Taylor and du Boulay (1986) where an overview of the problems with novices is provided.

This paper summarises the major findings already reported in relation to novices and also discusses the work done on expert performance. These observations are less complete than the work on novices, providing only a partial account of programming behaviour in the form of 'snapshots'. This data is presented below, but since the detailed discussion of procedures and experiments related to novice and non-novice Prolog learners are reported in Taylor (1987), only the conclusions are presented here.

After a short discussion about the study of novice programming, and a brief description of other related work on Prolog learners, the report is divided into three major sections.

Section 2 discusses some approaches to program writing used by Prolog expert users at Sussex University.

Section 3 discusses complete novices, and the difficulties they experienced during the early part of the learning process.

Section 4 reports on the problems encountered by a student who had used other programming languages as he tried to learn Prolog.

1.1 NOVICE PROGRAMMERS - Background remarks

Learning to program in any language is not an easy task, and teachers of programming will be very well aware of the myriad difficulties which beset beginners. The development of a new programming language is often followed by claims by its proponents that major problems associated with existing languages are on the brink of extinction. Predictably, though, new languages, whilst perhaps overcoming some particular types of difficulty, bring with them their own idiosyncracies with which learners have to grapple, and the fact remains that learning to program is a hard task, requiring dedication on the part of the learner and many hours of practice.

The literature on novice programming reflects several different perspectives in the analysis of programming performance. There are those concerned to model the acquisition of cognitive skill (Anderson, 1982), where the major emphasis is upon the psychological mechanisms which allow learning to take place, and which facilitate the development of skilled performance. These studies have been mainly undertaken studying students learning LISP. Others are more concerned to understand the misperceptions of novice programmers which result in buggy programs (e.g. Joni, Soloway, Goldman and Ehrlich, 1983; Spohrer, Pope, Lipman, Sack, Freiman, Littman, Johnson and Soloway, 1984). Such studies try to identify the characteristics of bugs, based on an analysis of the disparities between what the students think should happen, and what actually does happen in the programming session. Many of these bugs are associated with misinterpretation of parts of the syntax of the language, and misperceived relationships between its different components.

A related approach to understanding novice performance is to contrast it with that of experts, in an attempt to identify, and subsequently reduce, the distance between the two (e.g. Adelson 1984, Ehrlich and Soloway, 1982).

Other researchers have examined difficulties associated with successfully dealing with the complexity of the programming situation. This aspect has two distinct levels: that of complexity within programs, and their proper organisation; and that of complexity in the various modes and states of the computer itself. Soloway et al. (1984) discuss rules of programming discourse which help the learner understand how orchestrate various parts of the program to achieve a desired effect. du Boulay et al. (1981) discuss difficulties in relating the various 'notional machines' to one another, being aware of where one is in the system, and whom one is addressing (e.g. the editor, or the operating system, or the compiler).

Despite the enormous quantity of research on novice programming

a welter of support materials. The learner of a new language may be struggling with impoverished environments, insecure teaching, and a lack of supportive debugging tools. Some difficulties experienced by Prolog novices, therefore, may be as much a result of these factors as they are of genuine complexities in the language.

1.2 RESEARCH ON PROLOG NOVICES

Research into Prolog novice performance has had to move forward rapidly on a broad front. A brief annotated bibliography of the Prolog research community has been produced (du Boulay and Taylor, 1987). Reflecting some of the different approaches mentioned above, the major projects are as follows.

With regard to teaching issues, Bundy, Pain, Brna and Lynch (Edinburgh) are developing a coherent 'story' to tell Prolog novices. They identify seven partial models or 'stories' in elementary Prolog texts: OR trees, AND/OR trees, Byrd Boxes, arrow diagrams, flow of satisfaction, full traces and partial trees. However good these stories are for illustrating some particular feature of Prolog, Bundy et. al. criticise them for being ad hoc. Students often lack confidence to predict the behaviour of a previously unseen Prolog program because the stories do not mesh into a coherent whole. Bundy et. al. aim to derive a complete story that covers all aspects of Prolog in a uniform and coherent manner (Bundy, 1984; Bundy and Pain, 1985; Pain and Bundy, 1985; Bundy et al., 1986).

With regard to support tools, Coombs and Stell (Strathclyde) have investigated misconceptions of novice programmers with a view to building automatic debugging tools. Studies they have made of backtracking errors are used as a basis for protocol studies in the research reported here (Coombs and Stell, 1985). Ross (Edinburgh) has investigated both teaching issues as well as the design for a Prolog tutor (Ross, 1982; Ross, 1986).

Rajan (Open University), proceeding from the slogan that dynamic events require dynamic tracing, has developed a tracer which single-steps through code, highlighting relevant portions, and instantiating variables in the code in situ. He found that understanding of Prolog programs was greatly improved when the learner had access to the trace information (Rajan, 1985).

Eisenstadt and Brayshaw (Open University) have developed the Transparent Prolog Machine, which to some extent solves this problem. Using modern graphics workstations they are able to display an execution space of many thousands of nodes. It incorporates enhanced AND/OR trees which carry information about clause head matching, and provides a zoom facility to focus in detail on particular parts of the code. This system is aimed at experienced programmers, although it does allow for 'slow motion' tracing for those not so experienced. The system is still being developed and evaluated (Eisenstadt, 1984; Eisenstadt et al., 1984; Eisenstadt and Brayshaw, 1986).

Empirical psychological studies of Prolog learners have been conducted by Ormerod, Manktelow, Steward and Robson (Sunderland Polytechnic), who have investigated the effect of list

From the point of view of novices' errors, the work of Van Someren (Amsterdam) and the work reported here are complementary. Van Someren has investigated the 'mal-rules' novice programmers exhibit in constructing simple Prolog programs whereas we have tended to concentrate on the issue of the novice's interpretation of the programming task as construed in Prolog and as affected by their prior knowledge (Van Someren, 1984; Van Someren, 1985).

Our own work focuses on the very high-level misperceptions that novices have of the programming task, and the interaction between these and their interpretation of Prolog when presented with either the declarative or the procedural view. These more general misunderstandings can give rise to lower-level bugs of the sort discussed by Van Someren.

Some of these groups have recently contributed to a SERC/Alvey funded workshop on learning and using Prolog, and a collected series of papers is in preparation.

2.0 EXPERT PROLOG PROGRAMMERS

As part of the research project, we observed a group expert Prolog programmers. Before detailed discussion, some general points regarding the performance of these experts is in order.

Firstly, the study was not intended to be an analysis of expert programming performance per se. The object of the exercise was rather to get a feel for the approach and style of expert Prolog programming, to provide a measure against which to begin evaluating novice performance.

Secondly, because of the informal nature of the test materials, and the small size of subject pool, comments reported here are mainly anecdotal. Also, there was not a great deal in the way of uniformity of performance. However, there were certain characteristics which stood out, which are discussed below.

Thirdly, some of the questions posed in our original proposal were not addressed in our study. The questions pertaining to expert performance were:

1. Does programming in Prolog require (or encourage the development of) any specific kind of approach to problem solving?

2. Do experts sti3 3 make use of both the declarative and procedural semantics?

3. What is the Prolog folklore?

4. What heuristics and strategies have experts developed to cope with situations where they are unsure about how to proceed?

5. How good are experts at predicting what Prolog programs will do?

6. How do experts go about debugging?

Because of lack of time we did not address question 3 (Prolog folkore), nor question 4 (the heuristics and strategies developed by experts to cope with situations where they were unable to proceed). A much larger, more detailed study would be required to adequately answer these questions. However, through discussion of the protocols, points relevant to the other questions will arise.

2.1 METHODS AND PROCEDURES

There is at Sussex University a resident pool of Prolog experts, either faculty, research fellows or graduate students, who have both taught and/or used Prolog for some time. Seven members of the group elected to participate in the study: three members of faculty, two research students, and two systems programmers. It turned out that one of the members of faculty, and one systems programmer were not in fact Prolog experts, although they had considerable knowledge about programming and the system on which they were working. These two, although their work was interesting, do not form part of the study.

The experts were provided with a range of problems (see Appendix 1 for transcripts) mostly taken from How to Solve it with Prolog (Coehlo et al. 1980). One problem stood out as particularly interesting - the Architect Problem - and most of the discussion below focuses around this. We also asked subjects to write a program to flatten lists in Prolog, and one protocol in particular is referred to. Lastly, they were asked to debug a buggy program without running it.

Because all these experts were very busy, it was impractical to ask them to take time out to participate in video-taped protocol studies. Instead, a method had to be devised which allowed them to do a task at times to suit themselves. The data was gathered by means of an automatic logging program (written by Roger Evans) written in POP-11 and running on the POPLOG system.

The logging program read the text of the problem file into the current file the expert was using, and prevented the consultation of on-line help materials and documentation. The reason for this being that if the experts "disappeared off" into documentation files, the logging program would be unable to keep track of where they went. The experts were asked to comment at times when they would normally have pursued on-line (or off-line) information.

The logging program trapped all input to the editor (VED) from the terminal, and stored it in a file, together with elapsed time between typing characters. Only differences of 3 seconds or more between keystrokes were recorded. A second program was used to replay the logfile by substituting the characters stored in the file for the normal keyboard input to the editor. The editor then behaved as it did for the expert when the logging took place. The playback program allowed the experimenter to suspend playback, with the option of single-stepping through keystrokes, or resuming continuous playback.

2.2 SPECIFIC OBSERVATIONS

2.2.1 From Problems to Specifications

The major observation was that progress through a problem seemed to have quite distinct phases, reflected in the comments made by experts as they worked.

Aside from the domain specific knowledge which experts clearly must have, expertise is also associated with an ability to understand how to integrate, manipulate and combine relevant information in the programming task. From our observations of these Prolog experts, it seems that there are fairly clear stages in the program writing process in which distinct types of knowledge are being utilised: that associated with the nature of problems in general, that associated with formal representations of problem solutions (e.g. as in a program), and that associated with the methods and techniques for achieving a given effect.

These different types of knowledge are kept separate, and whilst the expert may 'jump around' these domains of knowledge whilst considering how best to solve the problem, there is no confusion about the kind of knowledge which is relevant to what kind of question. An effective programmer, therefore, not only has the relevant information (or has potential access to such information), but also a facility for moving from one level, or type, of description to another and back. Involved in this process is an understanding of the effects of moving into a particular domain, what the constraints are, and how that affects the shape of the eventual solution.

Familiarity with a domain enables the expert to make decisions about the sorts of conceptual tools which need to be available for a problem to be successfully solved. These decisions may involve stepping outside the domain in question to create an appropriate environment which either caters for, or overcomes, the constraints inherent in that domain.

To clarify, the following scenario is a hypothetical expert based on our observations of real experts. The first phase of a problem solving session would consist of problem interpretation (i.e. abstracting the problem structure from the English description). This takes place in the real-world domain, and the expert is establishing whether or not he understands the problem correctly (or adequately). Parts of the problem statement are queried, and decisions are made about what kind of a problem it is, and what would consititute a solution. This process corresponds to the construction of a problem space, or a mental model.

During this process, certain kinds of decision will be taken

real world domain into (in this case) the formal mechanistic domain. Comments may be made about the kinds of objects/relations/processes that will be needed to solve the problem and note will be taken of functional relations of parts of the solution (e.g. 'I think I'm going to use fast-set-of in order to whip through the set of objects I will have collected...'). Having decided this, the next issue is: 'have I got everything I need available?'. If not the expert either needs to develop it, or to retreive it from the system, which may involve moving out of the formal mechanistic domain.

Finally, the solution would be expressed in the syntax of Prolog. It may be that original design decisions taken at higher levels are inappropriate or unnecessary, so a certain amount of movement up and down the domain framework might occur. In this phase the expert will be aware that some kind of a solution has been reached (i.e. the program actually 'works' in some sense) but needs testing with different kinds of data. Knowing what kind of data to use depends upon the ability to discriminate between different kinds of behaviour produced by the machine at the formal mechanistic level.

We shall now discuss the evidence to support this view of the programming process from our expert protocols mainly in relation to the Architect Problem reproduced below.

The Architect Problem

This problem provided some interesting data about expert performance. It had been selected because it seemed that the English description of the program belied the simplistic (even over-simplistic) program provided in How to Solve it with Prolog (p. 63).

> Write a program for designing an architectural unit obeying the following specifications:
>
> 1. There are two rooms
>
> 2. Each room has a window and an interior door
>
> 3. Rooms are connected by an interior door
>
> 4. One room also has an exterior door
>
> 5. A wall can have only one door or window
>
> 6. No window can face north
>
> 7. Windows cannot be on opposite sides of the unit

Most of the experts reacted to the loose formulation of this problem:

51 This is an unbelievable problem. Design an architects unit satisfying the following constraints !!!!!! You got to be kidding.

In fact I can't even imagine what you want here. Points - 7 are, in a sense, a design for a building. I co translate them into Prolog database entries, but that's very exciting. As far as I can see, DESIGNING a buildo satisfying these constraints means drawing a floor-pl and if I knew how to write a Prolog program that would c floor-plans for buildings given a set of constraints 1 these I'd be selling it for cash, not knocking it up in spare time to give you something to work on.

52 Wow! animals [another problem] was easy and I took mins on that.... good job I'm not in a hurry. Excuse while I have a little think.

53 Not at all clear what designing means. Is it check constraints on a design? What's the input to the progr What's the output?

54 This problem is almost ludicrously open-ended.

After this initial phase, however, the next step was interpret the statement, and formulate a much tigh description of what was to be done. There were a variety candidate methods and approaches to this task. S4 explicitly out his plan:

S4 For example - what does "to face North" mean? if we h all walls at right angles and one is at NE then does t count? I'll approach this in a simple way.

Here are the constraints:
rooms have four walls
all walls are at right angles
one wall goes north-south

Here is an interpretation:
1. I'll re-interpret this to make it easier! (hahaha!) "a wall can have only one door or window" means a we can have a door, a window or neither, but not both.

2. each room has a window and a door means a room has at most two doors. In fact, one room has two doors (one interior, one exterior) and the other has only one (interior). But each room can have any number of windows (one or more...)

and here we P*O:

can have an interior door
or
can have an exterior door
or
can have a picture

SI also clearly illustrated distinct stages in his protocol. His first comment was:

> After a certain amount of scribbling, scratching my head, staring out the window . . . I've decided that what I shall ATTEMPT to do is design a system which will tell me which constraints my current design doesn't satisfy. But I'm going to do the designing, not it.

The design decision had been taken - the program would ratify his architectural design rather than create one itself. Some of the constraints were then specified:

> Some definitions: four wall that meet make a room two walls meet if they have a common end Constraints translated into Prolog 1, 2, 3, 4, 5. I can't easily see how to translate the other two (particularly not the one about North facing; walls) so I shan't...

Having interpreted the problem, eliminating parts of the statement which he considered to be either too difficult, or infeasible, the next phase was to implement the necessary operators for such a solution. This provided the environment in which the plan could be implemented, signalled by the remark:

> I now have, I hope, operators that will enable me to specify the layout of rooms, and to find out which of my constraints are not satisfied (I probably haven't but we'll come to that). So what we do now, is specify the dimensions of some walls, connect them together into rooms, and add doors and windows, checking every now and then to see what constraints need satisfying.

The protocol then proceeded with implementations and checks to code up the solution.

2.2.2 Setting up Environments

The phase described here is the setting up of an environment within wl>ich to solve the problem, and seems to be a critical point in program creation. It appears to be the transition point between some form of 'generalised' problem solving and the program writing phase, where emphasis shifts from worrying about aspects of the problem to how to program it. These experts often appeared to have an extremely clear idea of what kinds of

predicate took, or what its name was.

For example, S2 redefined operator precedences in order to use 'is' and 'are' as infix operators in the Animal problem. The first part of his protocol consisted of explorations of the Prolog system to establish what facilities he could use within the logging program. He experimented with conjectures about how precedences work, and checked error messages to discover what was going wrong. The resulting program was extremely easy to read:

```
omnivore are mammal,
herbivore are mammal.
 .....
hi isa herbivore,
ol isa omnivore.
  ...etc
```

The program supports clean declarative interpretation which belies the preliminary work S2 had put in to create the environment in which he wanted to write his program. At one level, the decision to write a program which had a surface correspondence to a pure logical specification was a design decision taken early on. However, the implementation of the program involved intimate knowledge of the underlying ProJog machine.

Another example was that of S5 who discovered that the built-in predicate 'retract' did not function as he wanted. Having traced it with the spy mechanism, he commented:

```
they've implemented retract wrong -
or at least not the way I want it.
```

He then proceeded to redefine 'retract' which did precisely what he wanted,. This again confirms the notion that the expert knows exactly what functional role a set of clauses ought to play in the problem solution, and if the system doesn't provide such a facility, then one is built. This would be contrasted with the novice who would only be able to utilise what the system provides to construct a solution and who may thus be trying to bridge an impossibly wide gap.

As mentioned previously the program which logged the interaction did not allow consultation of on-line documentation by the experts (a constraint that would have been at odds with their normal mode of behaviour). Several experts commented that at some point they would either consult a textbook, or look at help or documentation files. Since they were not in a position to do either, they embarked on a trial and error sequence, but in an extremely informed manner, suggesting that they knew what

For example, a consideration in Prolog programming is where to put the data - i.e. Prolog allows the user to create programs which have data in the form of ground clauses in a database, or alternatively to build complex clauses which fish out relevant information by unification. The final decision will often rest upon how general a solution is required.

S5 experimented with several different program structures, commenting:

> The way things are going, half the constraints are in the structure (e.g. one window per wall) while the other half are going to be at top level (e.g. no north facing window). This is yukky - either I put them all in the structures (easier, but less general) or all at top level (messy but general).

Eventually he scrapped all his previous exploration, and wrote a program of compound datastructures accessed by procedures. Some of these procedures had constraints in them.

2.2.3 Errors

We found that experts were as likely to make certain kinds of slips as are novices - e.g. syntax errors, spelling mistakes, a wrongly ordered argument sequence etc.

For example, mixing up the order of arguments in rules - this error was committed by S5, an extremely proficient Prolog programmer, who, after tracing the program to discover what was going wrong, ruefully remarked:

> The arguments were backwards - I knew I should have used infix operators to make sure I knew which way round things were..

His version was:

```
is_a(X,Y):- kindof(Y,Z), isa(X,Z).
```

which should have been:

```
isa_a(X,Y):- kindof(X,Z), isa(Z,Y).
```

Again, misspellings cause problems in Prolog programs, and one expert was floored by one. S1 used the trace to see how his program checked constraints. Given the input 'make_room(w1,w2,w3,w4)' he left spypoints set and checked through 92 lines of trace information to verify the design. However, he did not notice that in one of his predicates he had misspelt 'constraint' as 'constrant'. The program failed, and he

defined two predicates with the same name but each one took a different number of arguments. Consequently, the program ran quite happily when the predicate which was supposed to take three arguments was only given two, since there was a predicate of that name which took two. Unfortunately, of course, the solution was incorrect. The first important point was that the expert was aware that something could be wrong with a running program - the effect in this case was to produce too many solutions. S6 used spy set on 17 clauses, and after careful analysis of lengthy output he spotted his error.

Predictably, experts were prepared to dedicate a fair amount of time to systematically checking the output of the tracing mechanisms, a task which is liable to daunt many novices. But what emerges from the protocols is that experts were skilful at interpreting debugging traces, and were not prepared to leave much to cheerful optimism. When the debugging trace began to spew out, experts knew what they are looking for, and were prepared to study it for considerable amounts of time. The debugging trace referred to the execution/procedural semantics, but we propose that experts were not simply inspecting that information at face value (otherwise S1 would have spotted his spelling mistake). The process involved interpreting the output in terms of what that meant for the other levels of the problem solving process - it was interpreted functionally, not literally.

2.2.4 Use of trace information

We noted two different uses of the debugging traces. One obvious use was for 'simple' debugging. We expected to see experts debugging programs carefully and systematically, and certainly the protocols contained a great deal of trace information. However, it became clear that this trace information was not simply switched on after the program was written in order to check it out. Rather, the program was developed incrementally, by writing a piece of code, and then running the spy facility to check that it really was doing what the programmer thought it should, and as a means of illustrating to the programmer which cases the piece of code was not dealing with. A further use was to get an overall view of what the program was up to 'behind the scenes' as it were. There may be several reasons for this.

Firstly, Prolog's backtracking mechanism needs constraining. We have observed novices whose programs run away with them due to unconstrained backtracking. The experts were not only be aware of the pitfalls of unconstrained backtracking, but were also likely to put a premium on efficiency - it was not always sufficient simply to produce the working program, it was more challenging to do it in machine efficient terms.

was given in a matter of 10 minutes, whereas the experts took on average about 40 minutes. However, their solutions would cope with all kinds of lists (containing any type of structure) and could work efficiently.

The solution provided by S1 for this problem demonstrates the point. He made use of the debugging trace to successively refine the solution from his original stab:

```
flatten([X|L], [Y|K]):-
        !, flatten(X,Y), flatten(L,K).

flatten(X,X).
```

to the final version:

```
flatten([[X]|L1], L2):-
    flatten([X|L1], L2).

flatten([[X|L1]|L2], L):-
    flatten([X, L1|L2], L).

flatten([[]|L1], L2):-
    flatten(L1, L2).

flatten([X|L1], [X|L2]):-
    flatten(L1, L2).

flatten(X,X).
```

Five versions were created, and between each one was a large section of debugging trace. The subject examined the trace closely, and the next version was developed to cope with successively more cases.

The problem could have been solved straightforwardly by using 'append', but that version of 'flatten' is costly in terms of machine operations. All the other experts used append, but this last solution represents a version of flatten which does not.

The importance of observing the machine's behaviour to check that the program was correct (even though it may look correct) was emphasised by the fact that none of the experts in this study were willing to predict what a program would do without running it. This may only be a reflection of 'house style' at Sussex, but it may also be relevant to the deeper issue of understanding the overall function of a program. Prediction of the entire behaviour of the computer from a written program is not only difficult (i.e. one would have to mentally simulate complex sequences of execution) but may also be fruitless due to the high possibility of error. Unless they could inspect the behaviour of the computer whilst the program was running, our

behavioural component for evaluation purposes, the novice may find it difficult to know whether or not a logical specification is correct (see Taylor 1987, Chapter 3).

There are two remaining questions so far unaddressed for which we have only sketchy answers.

1. Does programming in Prolog require (or encourage the development of) any specific kind of approach to problem solving?

Many of the problems presented to the experts in our study focused on writing programs which satisfied constraints, and in this enterprise, heavy use was made of unification (or matching). The major issue in this type of programming was where to put the data, and how to access it. S5 provided an example of an expert debating what kind of approach to adopt (see above) and attempted several strategies. But this sort of approach used mainly in association with unfamiliar problems (e.g. the Architect problem) is as much linked to specifying the problem clearly as it is to Prolog programming. More routine (and familiar) exercises (such as the Flatten exercise) tended not to be planned this way, but rather were created by trial and error using the debugging trace.

It seems intuitively obvious that experts would be inclined to construct a mental model of the problem with the constraints of the language taken into account - i.e. it would not be very expert to devise a solution which is incapable of being expressed in the language, and not discovering this until the coding phase. It was interesting to note that our two non-experts (i.e. they were not expert in Prolog but were expert in other languages), who were excluded from the study, both failed to come to terms at all with the Architect problem, presumably being unable to match their specification with Prolog constructs.

2. Do experts still make use of both the declarative and procedural semantics?

This question led to much debate amongst the experts in the study. The strong argument for declarative programming is that program statments need not contain any procedural information at all - the machine should provide this interpretation. However, given that Prolog is not a pure logic programming language (i.e. cannot be fully understood without reference to the underlying execution), a weaker argument to support declarative programming suggests that it is useful to separate declarative information from control information. This argument was presented by S6 who defended the use of declarative interpretations on the grounds that 'the logical component can help you understand/express yourself in the procedural one, and vice versa'. In his approach to the Architect problem, he began by specifying the goal statement:

What we want from the design in which walls are parts of which rooms, and which walls have which doors and windows in them.

The goal statement is something like this:

```
room(R1), room(R2), not(R1 = R2),
walls_of(R1, W1), walls_of(R2, W2), one_in_common(W1,W2),
and the restrictions on what can go where.
```

After having defined most of the sub-procedures for this statement, the predicate devised is as follows:

```
house([room1(W1), room2(W2), in_door(C),
                    windows(WW1, WW2), ex_door(Wex]) :-
    n_walls(W1,4), n_walls(W2,4), only_one_in_common(W1,W2,C),
    opposite_sides(C, W1, W2),
    non_north(WW1, W1), not(WW1=C), non_north(WW2,W2), not(WW2=C),
    not(opposite_sides(WW1,W1,WW2,W2)),
    append(W1,W2,Wused), member(Wex,Wused),
    not(member(Wex,[C, WW, WW2])).
```

This is the most clear cut example of declarative programming provided in the expert protocols. Whilst other experts eventually produced programs which could be viewed as declarative specifications, these were supported by the prior creation of tools and constructs with which to write the program. Building these facilities depended on a great deal of knowledge about the underlying execution processes. In other words, 'basic' Prolog was used to implement a higher-level environment to enable such a program to be written.

2.3 CONCLUSIONS

Having completed this small study of expert performance, and then having observed novices (see below), the major conclusion was that the two groups differed not simply because experts had a great deal of knowledge about Prolog, but because they understood the constraints of the formal domains in which they were working. In other words, they were able to adopt a variety of 'meta-level' stances from which to evaluate and criticise their approaches to problem solving and program writing without becoming confused, or losing track of where they were. They were able to navigate around a conceptual framework which novices simply did not have, a framework which structured their problem solving activities and program writing strategies. This kind of knowledge seemed more crucial to successful programming than an accurate recollection of detailed programming knowledge. Attention seemed to be focused on functional relations between processes in the program, and the distinction between different levels of description (e.g. from logical specification to Prolog code).

For this reason, the analysis of novice programming had to begin from another angle, examining the very early part of the learning process, to establish what kinds of conceptualisations of the task of programming, and learning Prolog, beginners had. This was a necessary pre-requisite before examining- the development of expertise in programming skill.

3.0 NOVICE PROLOG PROGRAMMING

The assumption in our study was that Prolog novices did not simply lack knowledge about Prolog - it was clear from our observations that experts sometimes lacked specific knowledge about Prolog, but, importantly, this did not severly hamper their work. Besides, many novices have available to them all the requisite Prolog-specific knowledge needed to write programs at a given level, either in text-books, or teaching materials. Furthermore, novices do not come to the programming situation devoid of general purpose learning, or problem-solving strategies.

The difficulties seemed to stem from two main sources: first, a lack of understanding about how and when to use particular pieces of knowledge in the programming session; second, the unlearning of inappropriate, but powerful, general purpose reasoning or problem solving techniques which they already possessed. In other words, as many teachers are aware, the novice is not an empty vessel waiting to be filled up with correct information.

Taylor (1987) argues that in many studies of novice programmers several major factors are ignored: the character of general purpose, domain independent problem solving or 'understanding' strategies in combination with the learner's previous background, experience, and intuitive interpretations. Studies which divorce problem solving strategies from their subject consider how such strategies work when applied only to correct information. However, learners rarely have only correct information available. A prerequisite to effective problem solving is the ability to correctly interpret the problem statement according to the constraints in the domain, and knowing what is or is not relevant to its solution. Typical learners cannot automatically be expected to have this knowledge, and not unnaturally, they support their as yet weak problem solving methods by introducing information from previous experience, guesswork and intuitions. This alters the character of novice problem solving, making it not only different from that of experts in terms of speed and accuracy, but also in terms of what they think the problem is. The learner's conceptualisation and initial representation of the problem is liable to be very different from what it should be, but it is this conceptualisation to which problem solving methods will be applied.

Many other studies (e.g. Anderson, 1982) focus on the learning mechanisms involved in cognitive skill acquistion, where the major emphasis is upon the psychological mechanisms which allow learning to take place, and which facilitate the development of skilled performance. In contrast, the study reported here

The main focus, then, is on identifying very high-level strategies for interpreting the programming domain, and for problem solving, which novices bring with them.

In order to pin down some of the particular difficulties associated with learning to program, a framework is used (Taylor, 1987) which identifies the various domains with which a Prolog programmer should, in theory, be familiar, in order to program successfully. The three domains represent different views which can be taken of problems, and their descriptions. They are labelled: the 'real world' domain, the formal logical domain, and the formal mechanistic domain. Each domain consititutes a frame of discourse (in effect, an environment), and has associated with appropriate reasoning strategies, a specification language, and means by which to evaluate the success of the specification.

The three domains are depicted as follows, and each is briefly described in the following sections:

Frame of Discourse	Components of discourse → REASONING STRATEGIES	SPECIFICATION LANGUAGE	METHOD OF VALIDATION
REAL WORLD	Ordinary (Practical)	Natural Language (imprecise)	Empirical Causal/Behavioural (Human)
LOGICAL	Analytic (Formal Logical)	Logical Expressions (explicit)	Hypothetical/ Syntactic (acausal/atemporal)
COMPUTATIONAL	Analytic (Formal) Mechanistic	Programming Language (explicit)	Empirical Causal/Behavioural (Computer)

The Domain Framework

The diagram is used to map out certain kinds of difficulties and potential errors that novice may fall into. There is no single correct route to the production of a running program, or specification, and the diagram is not prescriptive - i.e. program creation does not have to move from the real-world to the formal logical domain and then to the formal mechanistic. However, the ways in which programmers move around these various domains, and their ability to keep separate the various views of programs and problems will affect the eventual success with which they write programs.

3.1 PROGRAMMING DIFFICULTIES

We divide the difficulties associated with learning programming into six overlapping classes which can be mapped into each of these domains. Again, these classes should not be thought of as general stages in learning to program or particular stages in the process of producing a working program. Rather they represent views of the programming process at different levels. Each of these views needs to be elaborated and assimilated by the student in order to become expert:

Real World:
(i) General orientation to programming
(ii) Interpreting Problem Descriptions

Logical:
(iii) Using formal languages

Computational
(iv) Understanding the notional machines
(v) Using standard structures
(vi) Pragmatics

Each view is characterised and associated difficulties outlined below.

3.3.1 The Real World

The real world domain is the one which is familiar to us all, novices or otherwise, and is meant to represent the ordinary world. The environment of the real world is that of human discourse. In this domain, the problem-solver can use many strategies to solve problems, taking advantage of existing knowledge, prior experience, and known facts. Strategies for problem solving include induction, deduction, guessing, inference and so on. Any type of information could in principle be used: e.g. visual information, recalled information, information from one's colleagues, analogy, beliefs, prejudices and so on.

The language for expressing problems and solutions is natural language, which is normally understood via ordinary comprehension processes (i.e. co-operative), using implicit inference to construe meaning where necessary.

At this level, the learner must understand how to take a problem description, couched in natural language, and abstract the important components of the problem - i.e. the objects and relations - in order to construct a mental model of the problem to be solved. Since the overall task is programming, we define the criterion for success as the extent to which another

So there is the general PROBLEM OF ORIENTATION, finding out what the task of programming is supposed to be, what it can be used for, what general kinds of problem can be tackled and what the eventual advantages might be of expending effort in learning the skill. It is worth pointing out that if beginners have had no prior experience with computing languages, Prolog appears to be a 'something and nothing' - i.e. they are unaware of the underlying 'machinery' which is working for them (the backtracking mechanism for example) and may find the examples provided either uninteresting or not worthy of the somewhat deeper consideration required to understand the principles involved.

There are also the difficulties of INTERPRETING PROBLEM DESCRIPTIONS. One of the skills that the beginning programmer has to master is that of reading a piece of text expressing a problem and deciding what that problem is. This requires an analysis of the major entities involved, of their relationships and how a solution may be obtained in principle. For some programming languages the kinds of entity about which problems can be stated are well delineated (e.g. numerical) and can be used as 'landmarks'. As Prolog allows statements to be made about any relationships and implications, there is no clear boundary between things that can be described in Prolog and those that cannot. One way to reduce this difficulty is to stress the notions of relationships and individuals and give the students lots of practice in using a given restricted vocabulary to express limited aspects of English sentences (see Ennals, 1984 for examples of this approach).

Even if the major entities and relationships are clear there is the problem of deciding how these should be represented. The relative freedom of expression which Prolog offers for representing information may hinder rather than help beginners. Methods of representation can have a profound effect on how easily a problem can be solved, and on how efficient the solution may eventually be.

A widespread problem that beginners face when interpreting problem descriptions is deciding how general a solution should be. This problem may occur in Prolog to a larger degree than in other languages for the reasons given above.

3.1.2 The Logical Domain

The logical domain differs from the real world in several important ways. Firstly, the range of strategies available within the domain are restricted, as are the methods of reasoning, and the data which can legally be brought to bear as part of the problem solution. The usual method of reasoning is deduction, and the information which is relevant is only that

means - i.e. in a non-cooperative way, using only explicit inferencing processes. In this case, the student must understand the constraints upon the domain, and understand how objects and relations may be represented.

This involves understanding how to express the information regarding the problem solution in the terms of expressions within the domain. Since most logical operations are syntactic, there is no behavioural component for evaluation purposes at this level.

These, then, are problems associated with the notation of the various FORMAL LANGUAGES that have to be learned, both mastering the syntax and their underlying semantics.

Logic programming languages - including Prolog - are purported to be higher-level than other more conventional languages because of their relationship to logic. Certain forms of logic have a long history of being used as problem solving tools, and in computing such forms are often used as specification languages. However, logical expressions are liable to be misconstrued by beginners unless they are forced into recognising the formal rules governing logical expressions, as distinct from the rules governing natural language expressions.

A manifestation of this type of misconception arises when some solution expressed in a program lies in understanding the meaning of the English, not in the logical structure of the program. Students can sometimes convince themselves that a program will work because it 'makes sense' to them in English.

3.1.3 The Computational Domain

The computational domain differs again from both the other two. Expressions in it are subject to strict interpretation according to what they make the machine do - i.e. the language has a functional semantics. Again the interepretation is analytic, and the problem solver must understand both the nature of the domain, and the way in which the language is used to effect action. The success of the expression of a problem solution can be evaluated according to the ability of another 'system' (in this case a computer) to correctly interpret the program, producing the desired behaviour.

There are also difficulties associated with the mapping from an understanding of the problem to an understanding of the general properties of the various machines that one is learning to control, THE NOTIONAL MACHINES.

Users will normally have to master not only the programming

Then there are difficulties in understanding the problem and in translating it into the terms of reference of the programming language, which can be hard if the concepts embodied in the language are entirely new to the student.

This area can present beginners with a great deal of difficulty because they have to understand, first of all, the computing system with which they are working, and then distinguish which elements of that system belong to Prolog, and which are the system's own. This calls for discrimination in interpreting error messages, and consistently maintaining the distinction between the Prolog program/database visible on the terminal screen, and the version of that program/database that Prolog has. Students frequently alter their programs and forget to 'reload' or 'reconsult' the new file; alternatively they inadvertently assert what are meant to be queries, thereby accidentally altering the program/database.

The general problem for beginners with Prolog is that the underlying notional machine is both powerful and complex with a surface behaviour that is hard to predict accurately. Prolog syntax does not offer clear pointers to what is happening 'behind the scenes'. We gave a simple program to students and asked them to predict what the machine would do with it. Most of them were capable of outlining one possible solution (the one they were expecting) but they gave incomplete descriptions of all the work the machine would have to do to achieve a solution. This lack of knowledge of the complex internal workings of the machine will make debugging particularly difficult.

Backtracking confuses beginers in other ways. We have confirmed the findings of Coombs and Stell (1985) where students have misconceptions about the order in which backtracking takes place.

Associated with notation are the difficulties of acquiring STANDARD STRUCTURES, cliches or plans that can be used to achieve small scale goals, such as traversing a list or transforming one structure into another.

We have not investigated the use of standard cliches in Prolog. However, we have noted interference effects from the inappropriate use of standard structures from other languages identified by Van Someren (1985).

Finally there is the issue of mastering the PRAGMATICS of programming - that is learning the skill of how to specify, develop, test and debug a program using whatever tools are available.

One of the findings from our work with experts was their much

novices, on the other hand, seemed to want to undertake this hard predictive task for themselves without help.

None of these six issues are entirely separable from the others and much of the shock of the first few encounters between the learner and the system are compounded by the student's attempt to dea] with all these different leveJs of difficulty at once.

In the research reported here, we have not been able to address all of these issues in detail. For example, we have not focused on standard structures and cliches, nor specifically on the pragmatics of programming. It seemed from our initial observations of novices that they were not sufficiently weJl oriented toward the task of programming for such a detailed analysis to be feasible. The major areas of interest, therefore, were those associated with the first impressions that novices, and experts in other languages, had of Prolog, and the kinds of interpretations they put on expressions in the language, and on the machine's behaviour. Because of this, the study of expert performance stands apart somewhat from the rest of the research because those subjects, by definition, were oriented to the task of programming.

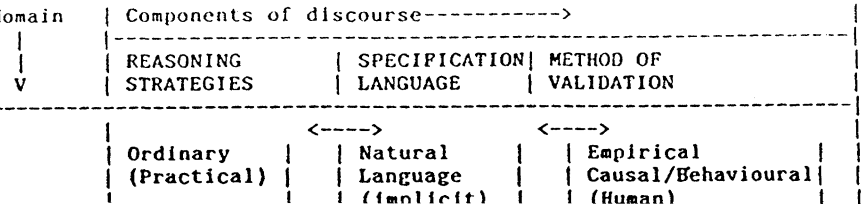
3.2 BUGS AND SUPERBUGS

Taylor (1987) uses the diagram sketched above to map out the space of possible errors that beginning Prolog programmers may have. Pea (1986) discusses 'superbugs' - high-level, language independent conceptual bugs - which disrupt the ways in which novices program and understand programs. One superbug identified by him is where beginners have the guiding analogy of human discourse for conducting their interaction with the computer. This amounts to assuming that inside the computer is a hidden mind with interpretative powers which can understand the intentions of the programmer. This analogy is often unconsciously adopted, since most learners will readily agree that there is not 'really' a mind inside the machine - however, they continue to act as though there were.

The specific problems which arise for beginners using this analogy are associated with the ways in which formal languages (such as a programming language) violate expectations about human discourse interpretation.

In effect, Pea's view of novices is that they are interpreting problems and programs from the real-world domain, and are implicitly assuming a co-operative understanding process on the part of the machine. This means that formal expressions are interpreted according to the rules of natural language, rather than in terms of the appropriate underlying models of those languages which provide their proper semantics (e.g. in the case of logic, perhaps a model theoretic interpretation; in the case of a programming language, the mechanistic rules by which expressions are interpreted). But not only will the language itself be understood by reference to natural language, the behaviour of the machine may also be interpreted according to criteria associated with understanding human behaviour. This may lead to the attribution of rather more intentionality to the computer than is warranted, and subsequent misunderstanding of the source of bugs in programs.

Taylor maps out the potential error sources, defining 'superbugs' as those which involve the use of interpretative processes appropriate to one level to interpret another (in the diagram the downward arrows). Bugs are defined as 'lateral' confusions - i.e. mistakes occurring within a domain rather than between domains. The diagram is as follows:



In the analysis of student performance, care must be taken not to assume that an apparent 'bug' is not in fact a symptom of a superbug. So, for example, a bug might arise where a student understands something about the constraints which govern the logical domain, has identified the appropriate objects and relationships necessary for problem solving, but has bad]y constructed the syntax of the logical expressions. Similarly, a student may have correctly formulated a program in the computational domain, but has not interpreted the behaviour of the machine correctly (perhaps obtaining unexpected, but correct, output) and begins to change the program.

On the other hand, a superbug is one where expressions in logic are constructed with a natural language 'meaning' hallucinated onto them - i.e. the intended meaning does not exist, or where the interpretation of the computer's behaviour is based upon the assumption that it can do more than it really can.

In order to establish whether or not a given error is a bug, or a symptom of a superbug, it is necessary to look at the context in which the error arose - i.e. careful attention needs to be paid to the type of language used by the student, and the means by which the student evaluated what happened and why. At this stage of the research, therefore, individual protocols yield the best data about superbugs, rather than group testing and statistical analyses (see below).

The major claim is that beginners who have no domain-specific -knowledge are obliged to INTERPRET a new domain. In this enterprise learners will make use of high-level strategies (of the sort to be found in the real-world domain) in the formal domains. Some of these strategies are not only powerful and economical when they work, but may equally introduce misconceptions and misinterpretations. In other words, what may turn out to be a superbug at some point in the learning process frequently started out as an essential and powerful general purpose learning strategy. As they apply these strategies, learners will bring to bear any existing information which seems to them to be relevant. This information will be woven into a 'theory[1] or a set of explanations by learners, which constitutes their growing model of the language. The situation is complex, therefore, because learners cannot be relied upon to introduce only correct information (as presented in text-books and teaching materials) into their conceptualisations of the programming language.

Lewis and Mack (1982), in a study of non-programmers learning to use a text editor, explore the issue of ad hoc explanations generated by learners to account for machine behaviour. They noted that learners who had little prior knowledge or current information to understand what was happening were able to

the first phase of reasoning in the development of new knowledge because it generates the hypothesis from which further reasoning (involving induction and deduction) can in principle proceed. Unfortunately, most beginners are content to accept the abductions without testing them out. An example of abduction is given:

> A learner was attempting to enter a password when a typing mistake caused the system to halt awaiting a correction. An indicator light marked 'input inhibited' came on. The learner attributed both the delay and the light to a heavy work load on the system.

This kind of explanation involves introducing a 'space of discourse' which contains new elements (i.e. the notion of work load) and these elements now form part of the learner's conception of the system. Abductive processes can override other forms of comprehension, even explicit statements in text manuals - learners sometimes re-interpreted instructions to say what they thought they should say, or discarded them as 'obviously' incorrect.

This process makes teaching difficult because learners may wish to defend their interpretations on the grounds that they have worked 'so far' - even though this itself is also a misperception on the part of the novice. Previous success may have been co-incidental (i.e. due to a peculiar set of circumstances), or it may be that learners believe themselves to have succeeded, when in fact they had not. But simply providing correct information is unlikely to remedy the situation, since it stands the chance, like original teaching materials, of simply being ignored, or of being re-interpreted to fit the student's model.

Taylor argues that there are two strands of difficulty in superbug effects for Prolog: that associated with interpreting the language, and that associated with interpreting the behaviour of the machine. These two types of difficulties are linked with the teaching approach taken. Students who are introduced to the declarative interpretation of Prolog, with its lack of emphasis on the mechanisms of the language, are liable to view Prolog as a cut-down version of English. This will lead to students embarking on a simple surface 'translation' exercise when writing programs, rather than a transformation of the problem statement. Such an approach results in underspecified programs because inference processes are left implicit in program statements, rather than being made explicit.

Students introduced to the procedural semantics, on the other hand, are less likely to make this type of 'translation' error, but may begin to interpret the execution strategies of Prolog as more intelligent than they really are. Assumptions about what

mtal models of the system. Instead, we focused on the kterpretation that learners had of the language and its motions in the very early part of the learning process. As has >en pointed out, many of these interpretations were misguided nee they were based on a real-world view of programming, ither than an appropriate formal domain view. This has adverse ?fects on the ability of the learner to adequately specify 'oblem solutions to produce effective programs. If learners are jsuming that human/computer interaction is analogous with unan/human discourse, then presumably a large part of the ^stem's workings are designated 'magical[1]. This would lead to le kinds of intentional language frequently used to describe **rolog's** workings - e.g. 'it just knows when to stop[1].

3.3 PROBLEMS WITH NATURAL LANGUAGE

3.3.1 Novices: Computers and Thought Students

The first group used to investigate problems with language were drawn from a first-year undergraduate course called 'Computers and Thought'. These subjects had no previous background in computing, and had not learnt any other programming languages. The course was an unassessed introduction to Artificial Intelligence, its approach and its techniques. It was not specifically a programming course, although students were introduced to Prolog. Programming assignments, however, were not graded on the quality of code, but rather on the appropriateness of program comments and discussion. These students were not expected to get to grips with many of the detailed executional aspects of Prolog, but were exposed to a broadly declarative view of Prolog applied mainly to database accessing and route-finding problems. However, they were not specifically taught logic, but concentrated instead on 'hands-on' experience of declarative programming.

Because of their declarative introduction, studies of this group centred mainly around the interference of natural language in interpreting and creating Prolog clauses.

This group was asked to perform two straightforward tasks: the first was to give an English rendering of some Prolog clauses, the second was to create a Prolog database from information expressed in English. Whilst the first task presented no difficulties, the second was problematic. The English sentences were a mixture of facts and conjunctions, such as:

John works hard and likes music

which are easily represented in Prolog, and some causally dependent sentences such as

John is a good student because he works hard

which is not so easy to represent, because Prolog does not handle causality directly. An elementary but adequate solution to this problem is to create a rule which says : infer that John is a good student if you can prove that he works hard. Some students were very resistant to this change of 'because' to 'if', which, in itself is not very surprising - the task is actually quite difficult. But some students made the change spontaneously, whilst others said that they could not resolve the difficulty because 'it doesn't mean the same'. One interpretation of this is that these students had not yet made the necessary break from the rules of natural language to the rules of Prolog expressions, where it is

The evidence obtained from the empirical studies supports quite strongly the view that making Prolog resemble natural language makes the learner's task more difficult. Interpolation of real-world knowledge and assumptions is frequently done unconsciously, and this misperception can be encouraged by the use of English in the programming domain. It would be interesting to establish whether this is true of non-English speaking novices, whose natural language syntax is very different from English (e.g. Japanese learners). A recommendation is that students be introduced to a diagrammatic representation of Prolog (e.g. the AND/OR trees of Bundy and Pain) rather than depending upon the student to distinguish between understanding arising out of natural language interpretation and understanding arising from an appreciation of the logical structure underpinning Prolog clauses.

Because of the problems indicated above, we found that beginners sometimes became very confused as they tried to map from one domain to another. It was found that rendering Prolog clauses into English presented few problems, but going from English to Prolog was fraught with difficulty. The situation is compounded by Prolog's inscrutable syntax, which allows learners to hallucinate spurious meanings onto syntax objects: e.g. interpreting the comma used as a separator between arguments to mean 'of'.

One of the main points about novices' difficulties in learning to program is that programming languages are formal languages, strictly interpreted according to rules. Many beginners clearly have trouble in identifying the limitations of formal domains, and the strictures which these put on potential problem solutions. Since this seems to be a source of many difficulties for a sub-set of learners of Prolog, there is no reason to suppose that teaching them logic beforehand will overcome the problem, since it too is an example of a formal language. That is to say, for those learners who find formal domains difficult, it matters little whether it is logic or Prolog. The superbug of interpreting expressions as natural language will remain.

Furthermore, since Prolog is not a pure logic programming language, there is a risk of confusing matters even more as learners attempt to distinguish between pure logic and 'Prolog logic'. This is not to suggest that accomplished logicians would not find Prolog relatively easy to learn; the question is rather whether it is a better teaching strategy for a genuine novice. The ability of novice programmers who are also logicians to cope with Prolog has yet to be empirically investigated.

3.4 PROBLEMS WITH INTERPRETING MACHINE BEHAVIOUR

3.4.1 Novices: MSc students

This study was of MSc students describing the execution of programs. The main focus is on their understanding of Prolog's automatic mechanisms, in particular backtracking, and their ability to predict behaviour from program statements. The MSc was a conversion course from an Arts Bachelor's degree to a Science Master's degree. It was very different from Computers and Thought being an intensive course during which students were exposed to both POP-11 (Hardy, 1982) and Prolog. They were required to come to terms with real A.I. programming, and were expected to be able to write quite large programs. These students were introduced to the procedural semantics of Prolog. They also had lectures in logic and resolution as part of their knowledge representation course. In contrast to Computers and Thought students, this group had learnt another programming language, and had been using computers intensively for about a term prior to this study. However, they were all new to Prolog.

Because of this background, MSc students were unlikely to fall into the same kinds of errors as Computers and Thought students. Whereas in that study the issue was what students thought program statements meant, studies of the MSc group were directed more at predicting the behaviour of the machine - i.e. relating program statements to the activities of the computer. Originally these studies were stimulated by the report of backtracking misconceptions by Coombs and Stell (1985), and the intention was to re-run the experiment at Sussex using MSc students. A further experiment was then to be conducted with the inclusion of the 'cut' in one of the clauses to investigate the effect of backtracking misapprehensions on interpreting its role in execution.

However, the protocols obtained from the first of these studies provided a great deal of information about other kinds of misconceptions that students had about execution. This provided us with the opportunity to do a two-levelled analysis: in the first case to discuss the specific bugs (and others) identified by Coombs and Stell and in the second to provide a bug/superbug analysis.

3.4.2 BUGS

This section describes the findings from the backtracking exercise described by Coombs and Stell (1985).

```
Clause  1:  a(X):- b(X), c(X).
Clause  2:  b(X):- d(X), e(X).
Clause  3:  b(X):- f(X).
Clause  4:  d(1).
```

Clause 10: ?- a(X).

Coombs and SteJl's (1985) Backtracking Program

Coombs and Stell had identified 'try once and pass' (TOAP) and 'redo body from the left' (RBFL). TOAP involves trying only one set of instantiations in Clause 2, and then passing on to Clause 3 instead of re-satisfying the sub-goals d(X) and e(X). RBFL consists of re-satisfying the subgoals in Clause 2 starting with d(X) rather than attempting to redo the 'e[1] sub-goal first - i.e. backtracking is perceived to progress from left to right instead of right to left. The following errors were identified and are briefly discussed below:

Domain Specific bugs:

* **Try** once and pass
* Redo body from left
* Multiple values for variables
* Try once and pass and cut
* Redo body from left and cut
* Parallel execution
* Failure to retry

* Database bug

Each of these is briefly discussed below.

Multiple values

lOJl variables

The most common error of this type was where students, having got an instantiation of 1 for d(X) in Clause 3, seemed to forget that this immediately would instantiate e(X) in the same clause to e(l). Therefore, at this point, the goal to be proved is e(l). Typically students thought the goal was still the more genera] e(X).

The Database Bug

It was pointed out earlier that the students had learnt POP-11 in the previous term, so some intereference effects were to be expected. The most obvious of these was the database bug. In this bug, the student appears to treat the Prolog database as analogous to the POP-11 database. A brief description of the differences between the two will help illustrate the bug.

Although the term *database[1] is often applied to Prolog programs, it has a slightly different connotation from when it **is** applied to databases in other languages. In many other **programming languages, databases contain data which** is **manipulated by externally defined procedures. Although in POP-11 it is quite feasible to simulate the structure of a Prolog-style database, the database itself is a global variable** [illegible]

POP-11 cannot 'do' anything, except be used for matching purposes by procedures. However, in Prolog, structures can act upon one another (e.g. by unification) and the compiler makes no significant distinction between facts and rules in this respect. The difference between facts and rules is not a functional difference, but the "POP-11 programmer" may regard them as corresponding to data and procedures.

This kind of misconception need not have devastating effects - it is usually the case that facts are fully instantiated, whereas rules are not, and as a meta-strategy on the part of the programmer, it can be useful to identify where possible instantiations are likely to come from - that is to say meta-level reasoning often stands one in good stead. Subjects were, after all, allowed to see the entire program throughout the session, and so it is quite reasonable for them to look at the facts to see from where instantiations could be derived.

But difficulties arise when students assume that Prolog can move around the program, consulting its facts and choosing its rules, in ways which it cannot, because this will disorientate students in terms of other facets of execution (e.g. backtracking).

Re-Running the Experiment with Cut

There is an exclusive relationship between TOAP and RBFL: if a student has TOAP then RBFL cannot be exhibited at the same time since the effect of TOAP is to eliminate the backtracking altogether. This, of course, has further ramifications for other Prolog constructs associated with backtracking, e.g. use of the cut.

To investigate the effect of these errors on interpretations of the 'cut', the study was re-run with a modified program which included the cut in Clause 2, and an extra clause to help prevent meta-level reasoning:

```
Clause 1:  a(X):- b(X), c(X).
Clause 2:  b(X):- d(X), !, e(X).
Clause 3:  b(X):- f(X).
Clause 4:  d(1).
Clause 5:  d(2).
Clause 6:  e(1).
Clause 7:  e(2).
Clause 8:  f(3).
Clause 9:  c(3).
Clause 10: c(4).
Clause 11: ?- a(X).
```

Modified backtracking task

In this case, a student exhibiting TOAP will feel no effect of

forward execution, the cut is automatically satisfied, but that one cannot backtrack through it, the RBFL error means that 're-doing[1] goals proceeds from left to right. Therefore, execution never does go 'backwards' through the cut. In these cases we expect to see students use Clause 3 to satisfy b(X), which should in fact be frozen out by the cut.

Try Once and Pjjsjs sLQ^ Cut

The above test actually caused a student who had not shown TOAP in the first experiment to produce it in the second. The actual misperception was to think that the effect of the cut was local to the clause - i.e. no backtracking could take place - but he then went on to try Clause 3. This clause should have been frozen by the cut.

B§I*2 Body J[rjqm the Lef t aj\d Cut

Similarly, RBFL arose for a student who had not exhibited it in the previous session. She began well:

1 Ok, uhm, first thing is er
2 look for a rule mentioning a(X)
3 and find one here
[indicating Clause 1]
4 and the first thing I have to prove is b(X)
5 I look for a rule telling me how to prove b(X).
6 I find this one,
7 first thing it tells me to do is look for d(X)
8 so I look for a rule mentioning d(X)

Having succeeded with X bound to 1, she looked for c(l), which failed, which, in this case, brought her back to the 'same rule' (line 34), and an attempt to re-do d(X) (line 36):

28 So I fail at this point here
29 with X as 1...uhm..
30 Kind of compressed into that..
31 looking for another c
32 there isn't one,
33 so I need to find another instantiation for b,
34 come to the same rule again
[indie. Clause 2]
35 to prove b,
36 I have to prove this, do d(X) first thing...

Because she concentrated on re-satisfying d(X), rather than re-trying e(l), she lost the emphasis on backtracking. She moved **left to right all the time, and** in **that sense, never did backtrack through the cut - she was always moving forward.**

'ParaLIPI' Expnntinn

of 1 for X, and was searching for e(1):

Subject 12

16 Then it looks to database to see if e(1) is there
17 and finds fact 6, that it is,
18 and them uhm, this instantiates b(X)
19 and the clause to value 1....uhm
20 It then tries to prove, I think,
21 rather than backtrack
22 and trying to prove d and e again,
23 I think it then goes to try and prove c.
24 Looks to database to see if there's c(X)
25 and c is 3.

Line 21 indicates a sense of premature backtracking, which this example was avoided, and he continued more or l correctly. In his second protocol, where the cut was included set off again to prove the 'e' goal, but showed the multi values for variables bug (line 13):

10 so it has proved d(1)
11 and then goes on to prove the next goal -
12 the cut - which automatically succeeds
13 and then goes on to attempt to prove e(X)
14 In the database there are e(1) and e(2),
15 so two possible ways of succeeding.
16 It tries the one and then tries the other
17 It would then, were it not for the cut,
18 go back to try the alternative solution for d
19 which would be d(2).
20 The cut makes it bypass the
alternative solution for d
21 so it goes back out.
22 Its now got a proof for b(X)
23 its now got to prove c(X)...

The experimenter queried what the current value of X was:

24 Present instantiation of X is d(1) e(1)
25 uhm.. e(2) would fail because X is the argument
26 of both d, the functors d and e,
27 and d(1) has succeeded,
28 but cannot be retried.
29 So it has to be d(1), e(1).

He was right to suggest that d could not be retried (line 2 but whether he had the correct reasoning was unclear. It a appeared from what he said, that the only reason Prolog did try all these options at once was because of the cut (line 2 The next step might indicate the source of error. There were c clauses in the program, which he thought would be 'compar

33 we look in the database and have two alternatives
34 which are c(3) and c(4).
35 Uhm. . so it tries c(3) and er...
36 backtracks because 3 is not 1,
37 and backtracks and then tries er c(4)
38 arid again 4 is not 1 so that fails.
39 So a(X) fails.

but there was some some spurious backtracking involved (lines 36 -37). These clauses simply do not match c(l), so there was no backtracking at this point. But the process of comparing clauses resembles the method he had used earlier within Clause 2 - i.e. in this final stage of the execution, all possibilities will be searched for, but since they do not match, nothing happens. However in the previous case, a match was found, which stops further searching. The student seemed to think it carried on.

FaililE^ J*≥ 'retry[T]

The last, and most common, omission which all but one subject had was the attempt to resatisfy e(l) prior to backtracking into d(l). In one sense this is a minor slip, since in this program e(l) cannot be resatisfied. But when it was explained to students that the attempt would still be made, without exception, they all thought it was a 'stupid thing to do'. Admittedly, in the context of this program, this might be so. But nevertheless, in more complex programs, it may be that e(l) could be re-satisfied some other way, in which case, execution would proceed with the instantiation of 1. The interesting point, though, is the declaration that this is 'stupid[1]. But Prolog could not know that the attempt is doomed to failure. The judgment on the part of the students has a ring of meta-level reasoning about it, a topic which will be discussed in the next section.

3.4.3 SUPERBUGS

This section discusses superbugs where the focus of attention is upon high level, general strategies that students have which appear to dispose them towards certain sorts of errors. The bugs and superbugs identified are:

* Meta-level reasoning superbug
    * Identity bug
    * WishfuJ Thinking: bug

* Reading superbug
    * Left to right control flow
    * Leftmost salience for arguments bug

infer some overall behaviour in the execution domain using either real-world knowledge, or knowledge about formal domains. Upon occasion, they were correct, though they were not able to give a blow-by-blow account of the full execution process.

Meta-level reasoning can become troublesome, however, when the student makes assumptions about the execution domain on the basis of prior knowledge, or experience with another, perhaps similar, task, but in this case an intuitive leap is made from the practical domain to execution domain, and Prolog is assumed to have made the same leap. This can produce bugs where students confuse their own reasoning and interpretative strategies with those of Prolog. This is not a superficial error, and students get into it because, initially, before they can describe what Prolog does, they have to understand the program themselves. If they have multiple ways of viewing programs - e.g. declarative or procedural - they can begin one way, and inadvertently start slipping into the other, and then introduce a dose of their own 'common sense' to help glue up holes in their explanation.

For example, the following student began by interpreting the program as if she herself were Prolog, and almost immediately began to identify Prolog's strategies with her own. This allowed her to 'luckily' pick up an instantiation for X - an instantiation which would not have been found at this stage:

1 Ok well if I was Prolog - well being not
2 quite Prolog but also myself - uhm the first
3 thing that I'm doing is looking on the rules
4 for 'a' and well.. looking at these 3 rules
5 together looking for what is my shortest path
6 to an instantiation down here. Uhm..I think
7 if I were Prolog I'd just look at number 1 and
8 luckily that would immediately get me a 'c'
9 that's instantiated... so....so I would try
10 instantiating X to 3, putting 3 in here
[in Clause 1].

She was right to suggest Clause 1 would be used first (line 7) but X would not be instantiated to 3 at that point. She had confused herself with Prolog at this point. It was she who noticed that the only instantiation for c in the program was from Clause 9, but the suggestion was (line 7) that this was what Prolog had done. The experimenter tried to point out the misapprehension:

EXP: Now are you doing that as yourself or as Prolog?

The subject continued but failed to appreciate the significance of what the experimenter had said:

11 Oh, ah, yeah I see. Now I'm back to being Prolog.

In other words, she still had the spurious instantiation of 3 for X.

Wishful Thinking bug

In this bug, students made assumptions about Prolog's abilities which left large holes in their account of what was going on. In a verbal report this technique may be used to gloss over the fact that the student simply does not know what happens. But one student, for example, hoped that Prolog would just accept instantiations, or 'take them for granted' rather than have to prove remaining sub-goals. There are cases where it is more probable that wishful thinking is a veil for simple lack of knowledge - the student is not sure how Prolog does something, and hopes in vain that a sensible guess will carry them past the trickier bits.

The Reading Superbug

Another superbug which appeared to influence students is the 'reading' superbug. Discussion of this superbug is speculative, based partly on observations of students doing the backtracking exercise, and partly on the experience of helping students understand the execution of clauses. Many of their problems suggested that novices seem to introduce aspects of their reading skills into the programming domain. The unconscious expectation is that progress through a program will flow from top-to-bottom and left-to-right, following Western reading patterns. Again, as with the meta-level reasoning superbug, this may be harmless, because in some respects Prolog does move from left to right. But the tacit assumption is that control flow through a clause or through programs is always left to right.

Emphasis in teaching upon Prolog's search through the database can confirm this tacit assumption. However, this does not prepare students to accommodate backtracking, or using clauses in ways other than originally intended. The common element with meta-level reasoning superbugs is again that students seem to assume that Prolog 'does as they do'. But the net result, of course, is that Prolog is thought to move through programs in ways which it does not.

Left to Right Control Flow Bug

The first bug resulting from the reading superbug is the tacit assumption that control through a clause passes from left to right. As has been pointed out above, this expectation may be violated, when instantiations are given to the right-most arguments. In this case, instantiations will appear to be passed from right to left.

Different instantiations, however, also appear to produce very

has its rightmost argument instantiated, it is probably better named 'split[1], but that the clause structures need not change is somewhat counter-intuitive. In other words, the name 'append' is on3y relevant to the programmer's intention at a given time. The actual structures which follow the predicate name have a set of prescribed behaviours which will take place whatever the predicate name, and the precise function is determined by instantiations in the input at that time.

However, if a student has watched the 'append' clauses being constructed in the normal way, then she is justified in thinking that the word 'append[1] now means (to the computer) what the structures in the clauses define it to mean. This can combine with what might appear to be natural language bugs - when students attribute *meaning[1] to predicate names it is not a simple case of thinking that Prolog Understands[1] in some magical way the word 'append'. It is instead a quite reasonable belief on the part of the student that the term 'append' designates an operation which has been defined within the clauses, and which she views as an inherently left-to-right process. So when these clauses suddenly produce what appears to be radically different behaviour, and the only thing which need be changed is the predicate name, students can get very confused.

L§Xt~IL[l]2-§t SeQijenjje *12L* Arguments Bug

A very closely related bug is one which leads students to attribute salience to the leftmost arguments, ignoring the effects on other arguments. For example, in the case where Prolog identifies a pattern (as *in* Example 3) students are confused as to how Prolog manages to 'stop' when the pattern has been established. The clause which achieves this effect of stopping is:

```
append([], L, L).
```

The ancillary knowledge which the student must have is that this clause can fail (thereby allowing the recursion to continue) in two distinct ways. The first, most straightforward case, is if the first argument is not an empty list. The second way is if the second and third arguments are not 'matchable'. That is to say, the ancillary knowledge required to read this clause is that the two arguments L and L must be able to be matched exactly.

The mistaken reading which often dominates the students' interpretation is that when the empty list is reached, then the clause will succeed, no attention being paid to the instantiations of the second and third arguments. In the case of identifying a match within a list, students confuse the facts *that (a) at Aar'h l#»y**l nf thp rpriirsinn, the* firQi" araiimpnt will

Here the potential match on the last two arguments can clearly be seen, since 'mar' matches 'mar', and we have put a variable into the tail which will unify with the list '[apr, may]'.

Preoccupation with the left-most arguments seemed to be quite pervasive in many students' interpretations of clauses in programming surgeries, and they continually needed to be reminded to look at other places in the program to understand why clauses were failing, or were not producing the correct output.

3.5 DISCUSSION

A two-levelled view has been taken of these MSc students interpreting a program designed to investigate backtracking misconceptions. The first view identifies bugs within the formal mechanistic, programming domain. From this perspective, students performing the task of interpreting the program were shown to have several misapprehensions. Not only were 'try once and pass' and 'redo body from the left' exhibited, but students' descriptions showed signs of other misconceptions: multiple values for variables, the database bug, complications of RBFL and TOAP with the cut; parallel execution and failure to retry sub-goals.

But the further analysis which focused on superbug interference identified other bugs which, when viewed from the formal mechanistic domain only, could only appear as undifferentiated muddles. However, these higher level misperceptions do seem to have a structure - they are not random - and in fact could be seen to lend coherence to the concrete bugs found in the mechanistic view. For example, 'try once and pass' and 'redo body from the left' could themselves be interpreted as examples (a) of the reading superbug, and (b) of the left-most salience for arguments.

There are two major points arising from this part of the study. Firstly, whilst it is useful to identify the particular manifestations of errors, it is equally important to keep in mind the patterns of misconceptions which give rise to specific bugs. Students may be suffering from a syndrome rather than a single misapprehension. Unless the aetiology of bugs is discovered, it is likely that some students will receive inappropriate remedial assistance, which can leave them in a more confused state than before, since underlying problems have not been addressed. Even these students who are receiving a strong tutorial background in both logic and Prolog's execution strategies exhibited signs of quite serious misconceptions, and the protocols illustrate how easily students, who are otherwise quite competent, can begin to slip into errors.

Secondly, as has been emphasised, superbugs are often developed from strategies which at some stage in the learning processes are not only sensible, but may also be the only ones available to students. For example, meta-level reasoning is an extremely powerful learning strategy for many purposes, and it is only under certain circumstances that expectations derived from such a view are misleading. The problem of bugs from superbugs, therefore, is not simply a matter of the learner 'getting it wrong'. The question is whether or not languages, or teaching methods can be designed so that such intuitive methods can be fully exploited, and appropriately discarded. This point has been made by Bonar and Soloway (1982) who discuss how students

intuitions.

Many of these bugs bear resemblance to Pea's (3 986) intentionality bugs (where the computer is thought to have foresightedness) in that they all have a similar effect: students assume that Prolog has a sort of meta-level view of the execution process, and can do things which it cannot. In many cases, though, this is less to do with an assumption that Prolog really is 'intelligent' than it is to do with students having a composite view of the execution process which includes tacit assumptions, for example, derived from reading habits, prior knowledge, and what the student herself would do to solve the problem contained in the program.

Another facet of interpretation of clauses, and predictions for behaviour is that beginners often have no choice but to abductively create 'theories' about what 'should[1] happen. It must be borne in mind that Prolog's inscrutable syntax can sometimes allow users to persist in an idiosyncratic interpretation for rather longer than is healthy. Because no immediately obvious constraints are imposed upon interpretation, for example, students can hallucinate very many different interpretations upon it, none of which reflect with any accuracy what. Prolog would actually do.

4.0 A NON-NOVICE PROLOG LEARNER

It might: be thought that experts in other languages would av many of the sorts of errors described so far. However, problems experienced by a student who participated in longitudinal case study illustrated that this is not necessar the case.

Alex represents quite a different type of subject than those far discussed, because he had learnt other programming langus before. But his study is important for several reasons. First it illustrates the role that prior knowledge plays conceptualising a new domain, and the determination with whic 'paradigm' can be adhered to, in the face of contradict evidence. Secondly, many of the problems Alex experienced the reverse side of what genuine novices go through - beca Prolog is flexible, learners are not challenged in th interpretation, and misperceptions can persist for a very l time. Thirdly, the longer term effects that a fundamen misconception has on performance can be traced.

The methodology for the case study follows Anderson et (1984) Since this form of protocol gathering produces enorn quantities of data, the discussion in tins section is organi around the original questions we had about non-novice Pro learners.

4.1 ALEX: A CASE STUDY

Alex was a postgraduate student, recently arrived from New Zealand. As part of his undergraduate degree in New Zealand, Alex had submitted assignments using Pascal, Fortran and a list-based command language, CPMI. He had learnt BASIC more or less on his own, and had run a tutorial for psychology students on an Apple microcomputer. He had also worked with the City Council using a Cobol database enquiry system (Datatrieve).

Alex's main problem was that his previous experience with programming was with conventional instruction-oriented languages (i.e. Basic, Fortran) and, as might be expected, he tried to construct a view of the Prolog execution domain in which all operations were 'reducible' to instructional terms of the sort with which he was familiar. In other words, the building blocks he tried to use were inappropriate for Prolog.

The major superbug was the data/procedures where basically he believed procedures to be active, and data to be passive. This led to considerable difficulties in understanding how to organise his programs effectively, and several other misperceptions arose as a consequence.

The first obvious point is that the languages he had learnt are quite different from Prolog not only in their internal structure, but in the kinds of problems to which they can be easily applied. Alex had not yet had any exposure to AI programming, and his first brush with it was through Prolog. Therefore, some of his remarks are better viewed as general comments on the activity of AI programming rather than specific to Prolog.

Secondly, his comments indicated that he had learnt his languages for very specific purposes: for assignments, for teaching, for working. Alex seemed to fall into the category of user who is interested in 'process' or 'what can I do with it' rather than someone who had a theoretical interest in the principles of a language. He 'never learnt them rigorously' but wanted to get a 'feel' for them.

It is also worth noting that Alex referred to his previous background as 'scientific programming' which implied an entire approach and methodology. In this respect, he was not being asked simply to give up some concepts or change his view a little to accommodate Prolog. He was being asked to change his whole attitude, his strategies and methods in favour of what looked to him like a sloppy, undisciplined approach which was bound to be fraught with errors because it did not adhere to 'proper programming principles'.

The net result of these factors was that Alex appeared to be

him by providing a task which fitted into his conception of 'interesting[1].

There are several themes which run through Alex's protocols. The major theme is the way Alex perceived the relationship between data and procedures. Alex's formal introduction to scientific programming had cultivated the habit of keeping data and procedures separate. Alex stuck to the view that data is passive and procedures are active throughout the study period. However, such an interpretation had major ramifications for structuring programs, and in particular for structuring databases. Even as late as the last session, Alex found it hard to think of appropriate ways of structuring information, even though he had a template program which he was adapting.

The secondary themes, which contributed to the program organisation problem, were associated with Prolog's use of variables, and the use of lists in programs. Variables in Prolog behave very differently from those in other languages, and one might expect to see a certain amount of confusion initially. Similarly, although Alex believed himself to be experienced in list processing, the kinds of lists he was familiar with were lists used in languages like BASK', which have very different characteristics to lists used as data-structures in programming languages like Prolog.

Because of these difficulties, Alex found it difficult to conceptualise the 'real' Prolog. He seemed to be waiting for the curtain to be lifted on some core quality which would be comprehensible, and straightforward to understand. Consequently, various representations of Prolog used in debugging tools and trace packages which did not quite tell the 'whole truth' about Prolog, were momentarily mistaken for the 'real' Prolog, which caused some understandable confusion.

Alex's case study demonstrates that those students who have a background in computing, and who might be expected to be orientated towards the programming domain, can still get into difficulties which resemble the sorts of confusions that real novices experience. The basic strategies Alex used throughout his learning period were governed by his prior knowledge, and his commitment to a particular view, which happened to be inappropriate in this case. As noted earlier, there seems to be supporting evidence for Lewis and Mack's (1982) 'abducting learner' in his protocol, since Alex was highly selective about what he took notice of, and seemed to disregard explanations and tutorial assistance from either the experimenter or the text book. Such information as he did take in, he tended to convert into terms with which he was familiar. These resistant phases were punctuated, however, by insightful behaviour.

The f data **and** nrnrpdnrps' misnnnrpntinn **sf»pmpd to hf» at thp** rnnt

term 'list', the idea of using lists as high-level datastructures would probably not be appealing to him if he regarded data as 'passive' and felt that it was only 'procedures' that did anything interesting. The distinction, then, between data and procedures had a long-term effect on Alex's ability to write programs with appropriate structures.

We now consider a specific question set out in our original proposal.

Q: Is it an advantage or a disadvantage to have learnt another language before tackling PROLOG?

The answer to this question in this particular case is evidently: it is not an advantage. However, it may be that other AI languages such as LISP or POP-11 could provide a foundation for interpreting Prolog more straightforwardly. The major difficulty - in contrast with the MSc students who had learnt POP-11 - seemed to lie in Alex's attitude towards his own expertise. That is, he approached the learning situation defensively, and proved somewhat unwilling to cast off some of his previously learnt habits.

A further difficulty, though, is associated with the kinds of building blocks that his experience was constructed upon. This affected not only interpretation of programs written in Prolog, but also the approach to problem solving cultivated by other languages. It was noted in the study of experts above that they seemed to move between problem specification and language constructs as the program is designed. Unfortunately for Alex, most of his specifications did not fit easily into the Prolog ethos.

This transition may not be easy to make if the older habits have proved successful in the past. Furthermore, it can be hard for such a learner to recognise the parts of the new language which are truly similar to known concepts, and those which are very different. Several times during the study Alex ignored what was being said because he believed he already understood. However, it was clear that whilst he understood, for example, the logic of the operation in question, he was not recognising that the technique for implementing it was very different from his other languages.

The declarative semantics, in this case, served mainly to disguise these differences. Alex knew, for example, the logic underlying the 'member' relation, but his misperception of its implementation led him to believe that it was a procedure which would be called to act upon data (i.e. a kind of function). For some time he failed to recognise the matching process which underlies the Prolog version of 'member' and this failure undermined his understanding of recursion in Prolog.

4.2 CONCLUSIONS

We have discussed how a basic misconception in Alex's view Prolog had far-reaching effects on his subsequent learning have noted that he was committed to his 'scientific programmi and was, quite rightly, unwilling to simply give up th notions unless totally convinced that it would be worth while. Learning, in this case, was very clearly influenced previous experience. He introduced into the situation fact which had only limited relevance to the learning of Prolog, which were very important issues for him.

In this respect, he was as susceptible to superbug interfere as a real novice, although the source of superbug is fro lower level in the multi-levelled framework, rather than higher levels. The main point to bear in mind from t discussion is that such interference is not simply dealt with presentation of 'correct' information. Alex had a 'paradi which had served him well up till now, and which he would easily relinquish. This paradigm affected his interpretation Prolog, and of the problems which he was trying to solve.

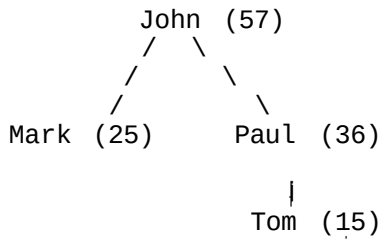
5.0 SUMMARY

These studies demonstrate the complexity of the task which the novice programmer faces. The sorts of errors that have been observed in Prolog learners have a range and depth that cannot be accounted for in simple terms. The multi-levelled framework has been used to distinguish superbugs from bugs, and to help orientate discussion, emphasising the fact that errors and misconceptions can have their origin in several ^sources. The main principle is that each of the domains which bear on Prolog are governed by rules and constraints which not only limit what can be done, but also govern what methods and techniques can be used in achieving an intention or goal. The 'meaning' of problems or programs within each domain is very different, and it has been shown that a major difficulty for novices is that they do not understand what the implications of moving from one domain to another are, and what constraints govern the methods available for problem representation or problem solution. Due to this lack of knowledge, powerful intuitive strategies are used, which affect the novice's perception of the problem to be solved, and consequently disorientate the program-writing process.

The major points arising are that; programming bugs can have their origins in very high level misconceptions, and they may show themselves in various guises. Furthermore, the same, or similar bug may be turned up by different students for quite different reasons. Superbug analysis is an essential part of understanding novice performance, and it is emphasised that a programming language's ease of use may depend crucially on the extent to which it nurtures superbugs.

PROBLEM 1:

Suppose we are given the following family tree, with e person's age in brackets:



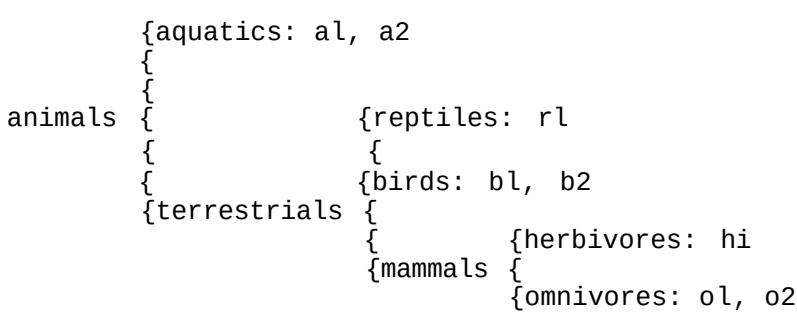
and we wish to find all solutions to the question "Does J have two descendants whose ages differ by ten years?"

PROBLEM 2:

Suppose you have a universe of eight animals:

U = [al, a2, rl, bl, b2, hl, ol, o2]

classified in the following way:



Write a datastructure representing this classification, build up a program for answering the following questions:

1) What is the classification of animal X?

2) What animals have classification Y?

PROBLEM 3:

Write a program to check whether a word is a palindrome.

PROBLEM 4:

Write a program for designing an architectural unit obeying following specifications:

1. There are two rooms

2. Each room has a window and an interior door

3. Rooms are connected by an interior door

4. One room also has an exterior door

5. A wall can have only one door or window

6. No window can face north

7. Windows cannot be on opposite sides of the unit

PROBLEM 5:

PLEASE DO NOT RUN THIS PROGRAM UNTIL YOU HAVE COMPLETED PART A.

PART A:

Given this program, can you (a) say what it does, and (b) WITHOUT RUNNING IT FIRST can you say whether it will work.

PART B:

After you have completed Part A you may run the program if you wish to check what you have said, making any amendments you want.

THE GIVEN PROGRAM:

```
begat(john, mark).
begat(john, paul).
begat(paul, tom).

is_aged(john, 57).
is_aged(mark, 25).
is_aged(paul, 36).
is_aged(tom, 15).

has_descendant(X,Z):- begat(X,Z).
has_descendant(X,Z):- has_descendant(X,Y),
                      begat(Y,Z).

find (X, Y) :- has_descendant(john, X), is_aged(X, N),
               has_descendant(john, Y), is_aged(Y, M),
               M is N + 10.
```

PROBLEM 6:

Given two lists, where the second one is the concatenation of a third one with the first one, find this third list and build up

population density per square mile, and find countrie
similar population density (differing by less than 5%).

```
pop(china, 825).
pop(india, 586).
pop(ussr, 252).
pop(usa, 212).
area(china, 3380).
area(india, 1139).
area(ussr, 8709).
area(usa_t 3609).
```

PROBLEM 8:

Please write a program to flatten lists.

PROBLEM 9:

Write a program for colouring any planar map with at most:
colours, such that no two adjacent regions have the same col

PROBLEM 10:

Can you say what this program does? Can you say whether it wi work?

PLEASE DO NOT SPEND MORE THAN 10-15 MINUTES ON THJS PROBLEM

THE GIVEN PROGRAM:

```
simplify(L, NL) :- compact(L, LI), simplifyl(L3, L2),
                   simplify2(L2, NL).

compact([L, []], L).
compact([L], L).
compact([Ll, [L2]], L):- concatenate(Ll, L2, L).
compact([Ll, L2|LN], L):- concatenate(LI, L2, X),
                          compact([X, LN], L).

concatenate([X|Ll], L2, [X|L3]):- concatenate(Ll, L2f L3).
concatenate(f], L, L).

simplifyl([X|L], NL):- member(X,L), !, simplifyl(L, NL).
simplifyl([X|L], [X|NL]):- simp.lify3(L, NL).
simplifyKU, []).

simplify2(L, NL):- simp]ify3(L, LI), subtract(Lf LI, NL).

simplify3([X|L], NL):- compare(L, X, []), simplify3(L, NL).
simplify3([X|L], [Y|NL]):- compare(L, X, Y), simp]ify3(L, NL).
simplify3([], []).

compare(_, [], []):- !.
Compare([Y|L], X, [Z|NL]):- (linked(X, Y), Z = X ;
                             linked(Y,X), Z - Y),
                            compare(L, X, NL).
compare([Y|L], X, NL):~ compare(L, X, NL).
compare([]» _. [])•

subtract(L, []t L):- !.
subtract([HjT], L, U):- member(H,L), !, subtract(T, L, U).
subtract([H|T], L, [H|U]):- !, subtract(T, L, U).
subtract(_, _, []).

member(A, [A|_]).
member(A, [_|L]):~ member(A, L).
```

PROBLEM 11:

Define combinations of a list, where s(K) indicates tl successor of K. For example, 3 is written (s(s(s(0))).

REFERENCES

ADELSON, B., (1984). 'When Novices Surpass Experts: the difficulty of a task may increase with expertise', J. Exp. L§ZChoLo£y, k§M!llDJg' M20L2LY sUQSi Cognition, Vol J,0, No.5, 483 - 495.

ANDERSON, J.R., (1982). 'Acquisition of Cognitive Skill', Psychological Review Vol. 89, No.4, 369-406.

ANDERSON J.R., FARRELL R. and SAUERS R., (1984). 'Learning to program in Lisp[1], Cognitive Science, Vol 8, 87-129.

BONAR, J., and SOLOWAY, E., (1982). 'Uncovering Principles of Novice Programming[1], B§^§JLQ!l E§£2£t No. ^£0, Department of Computer Science, Yale University, Connecticut.

BUNDY, A., (1984) 'Simple Prolog Prototypes', work in progress report, University of Edinburgh.

BUNDY, A., and PAIN, H.f (1985) 'Evaluating Prolog Environments', work in progress report, University of Edinburgh.

BUNDY, A., PAIN, H., BRNA, P., and LYNCH, L., (1986). 'A Proposed Prolog Story' Department of Artificial Intelligence, University of Edinburgh.

COELHO, H., COTTA, J.C., and PEREIRA, L.M., (1980), 'How to Solve it with Prolog[1], Ministerio da Habitacao e Obras Publicas, Laboratorio Nacional de Engenharia Civil, Lisboa.

COOMBS, M. J., and STELL, J. G., (1985) 'A Model for Debugging Prolog by Symbolic Execution: The Separation of Specification and Procedure', Department of Computer Science, University of Strathclyde.

di SESSA, A., (1982). 'Unlearning Aristotelian Phsyics: A Study of Knowledge-Based Learning[1], Cognitive ScjLerice, Vol. 1, 37-75.

du BOULAY, B., O'SHEA, T. and MONK, J., (1981). fThe black box inside the glass box: presenting computing concepts to novices', Int. J. Man-Machine Studies, 14, 237-249.

EHRLICH, K. and SOLOWAY (1982). 'An Empirical Investigation of the Tacit Plan Knowledge in Programming', Department of Computer Science, Yale University, Connecticut.

EHRLICH, K., SOLOWAY, E., and ABBOTT, V., (1982). 'Transfer Effects from Programming to Algebra Word Problems: a preliminary study' Department of Computer Science, Yale University, Connecticut.

EISENSTADT, M., (1984), 'A Powerful Prolog Trace Package', Procs. 6th European Conference on AI, Pisa.

EISENSTADT, M., HASEMER, T., KRIWACZEK, F., (1984), 'An Improved User Interface for Prolog', Procs. INTERACT-84, IFIP Conference on Human Computer Interaction, London, 1984.

EISENSTADT, M., and BRAYSHAW, M., (1986). 'The Transparent Prolog Machine: An Execution Model and Graphical Debugger for Logic Programming', Technical Report No. 21, Human Cognition Research Laboratory, Milton Keynes.

ELSHOUT, J.J., JANSWEIJER, W.N.H., and WEILINGA, B.J., (1986). 'Modelling the Genuine Beginner' Laboratory of Psychology, University of Amsterdam.

ENNALS, R., (1984). Beginning Micro-Prolog, Ellis Horwood.

HARDY, S., (1982) 'The POPLOG Programming Environment' Cognitive Studies Research Paper, University of Sussex.

JONI, S., SOLOWAY, E., GOLDMAN, R., and EHRLICH, K., (1983) 'Just So Stories: How The Program Got That Bug' Procs. of the SIGCUE/SIGCAS Symposium on Computer Literacy, Baltimore, June 1983.

LEWIS, C., and MACK, R., (1982). 'The role of abduction in learning to use a computer system', Technical Report, No. RC9433 (41620), New York: IBM Thomas Watson Research Center.

ORMEROD, T.C., MANKTELOW, K.I., STEWARD, A.P., ROBSON, E.H., (1984a) 'Reasoning with Hierarchies Written in Prolog Form', Department of Mathematics and Computer Studies, Sunderland

PAIN, H., and BUNDY, A., (1985), 'What Stories Should We Tell Novice Prolog Programmers?', work in progress report, University of Edinburgh.

PEA, R. D., (1986). 'Language-Independent Conceptual "Bugs" in Novice Programming', J. Educational Computing Research, Vol.2, No. 1, 25-36.

PEIRCE, C., (1958). 'The logic of drawing history from ancient documents' in A. Burks (ed.) Collected Papers of Charles Sanders Peirce, Cambridge Mass.: Harvard University Press.

RAJAN, T.,(1985). 'APT: The Design of Animated Tracing Tools for Novice Programmers', Tech. Report No. 15, HCRL, The Open University, Milton Keynes.

ROSS, P., (1986). 'Some Thoughts on the Design of an Intelligent Teaching System for Prolog', Department of Artificial Intelligence, Edinburgh University.

ROSS, P., (1982). 'Teaching PROLOG to Undergraduates', in AISBQ, Autumn 1982, 16-17.

SHEIL, B. A., (1980). 'The Psychological Study of Programming', ACM Computing Surveys, Special Issue, Vol.13, No.1, 101-120.

SOLOWAY, E., (1986) 'Learning to program = learning to construct mechanisms and explanations', Comms. of ACM, Vol. 29, No. 9, 850-859.

SOLOWAY, E., LOCHHEAD, J., and CLEMENT, J., (1982). 'Does Computer Programming Enhance Problem Solving Ability?' in R. Seidel (ed.) Computer Literacy, New York: Academic Press.

SOLOWAY, E., EHRLICH, K., (1984). 'Empirical Studies of Programming Knowledge', in IEEE Transactions on Software Engineering, Sept. 1984.

SPOHRER, J., POPE, E., LIPMAN, M., SACK, W., FREIMAN, S., LITTMAN, D., JOHNSON, L. and SOLOWAY, E., (1984) 'Bugs in Novice Programs and Misconceptions in Novice Programmers' Department of Computer Science, Yale University, Connecticut.

TAYLOR, J. (1984) 'Why novices will find learning Prolog har
Proceedings ECAI, 1984

TAYLOR J. (1987) 'Programming in Prolog: an in-depth study
problems for beginners learning to program in Prolog', D.F
Thesis, Cognitive Studies Programme, University of Sussex.

TAYLOR J. and du BOULAY J.B.H., (1986) 'Why novices may 1
programming in Prolog hard', Cognitive Studies Research Pa
no. 60, Cognitive Studies Programme, University of Sussex.

VAN SOMEREN, M. W., (1984), 'Misconceptions of Beginning Pr
Programmers' Memorandum 30, Department of Experimer
Psychology, University of Amsterdam.

VAN SOMEREN, M., (1985) 'Beginners Problems in Learning Prol
Memo 54, Department of Social Science Informatics and Departi
of Experimental Psychology, University of Amsterdam.