Programming in PROLOG: an in-depth study of problems for beginners learning to program in PROLOG

Thesis

How to cite:

© 1988 The Author

Version: Version of Record

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online's data policy on reuse of materials please consult the policies page.
PROGRAMMING IN PROLOG:
An In-Depth Study of Problems for
Beginners Learning to Program in Prolog

JOSEPHINE ANNETTE TAYLOR

Submitted in accordance with the requirements for the degree of D. Phil

School of Cognitive Sciences
University of Sussex

May, 1988
Declaration page not scanned at the request of the University
Programming in Prolog: An In-Depth Study of Problems for Beginners Learning to Program in Prolog

Summary

The main aim of this thesis is to examine the difficulties that beginners have learning the logic-based programming language Prolog (Colmerauer, Kanoui, Pasero and Roussel, 1973). Logic programming languages, such as Prolog, are a recent development, and whilst it is claimed they are easier to use than other current programming languages (Kowalski, 1979) there are no empirical studies of novice programmers learning such languages.

Observational studies using video-taped protocols were undertaken of different groups of novice Prolog programmers: the first were undergraduate students who had been taught the broadly declarative approach to Prolog programming, the second were postgraduate students who had been taught the procedural view. A longitudinal case study of a postgraduate student learning Prolog was also conducted.

It was found that existing methods of analysing programming performance, whilst useful for understanding some parts of the skill acquisition process, assume that learners are already orientated towards programming as a formal skill, and understand the kinds of tasks they have to perform in the context of formal problem solving. Many of the students observed in this study, however, used high level domain independent reasoning processes in their interpretation of programming tasks, producing plausible, but often incorrect or inadequate, solutions. This behaviour seems to share common characteristics with subjects observed in the literature on hypothetico-deductive reasoning.

Using this literature and that on novices learning programming, a three-levelled discourse framework of the domains relevant to programming is presented and is used to help map out the structure of the space of errors for these Prolog programmers. Transition from one discourse level to another requires a correct and consistent mapping. The empirical studies are used to show how novices construe Prolog using powerful intuitive strategies. These 'superstrategies' are essential for working in unfamiliar domains, but their use can equally lead to misapprehension and error - 'superbugs'.
DEDICATION

I would like to dedicate this thesis to my son, Alexander, without whose help and co-operation it could never have been written. He has yet to discover what his mother is like when she is not 'writing her thesis'.

ACKNOWLEDGEMENTS

My first thanks go to my thesis supervisor, Ben du Boulay, whose patience and endurance have been severely tried, but who managed nevertheless to remain cheerful, optimistic and constructive even in the darkest moments. His help, support and friendship are greatly valued.

For stimulating discussion and advice, I would like to thank: Andrew Law, Mark Elsom-Cook, Jonathan Laventhal, Claire O'Malley, Stevie Draper, Manny Rayner, Aaron Sloman, Maarten van Someren, Tom Khabaza, Roger Evans and Yvonne Waern.

My particular thanks go to Roger Evans for kindly providing programming support on demand; to Tom Khabaza for instructive discussions about Prolog; and to Claire O’Malley for detailed and thorough comments on both the content and the presentation of the thesis.

I would like to thank Judith Dennison, not only for keeping computing facilities up and running, but also for continued solace and support.

My grateful thanks also go to all those long-suffering anonymous students who co-operated so willingly in my studies.

Finally, my greatest debt of gratitude is to Andrew Law, whose staunch support, encouragement, and practical assistance in so many different ways made the process of writing up as painless as it could possibly be.

The research was supported in its initial stages by GEC Research Laboratories.

Note: In accordance with the practice of my elders and betters, I use the pronoun 'she' in place of 'he'.
## CONTENTS

### CHAPTER 1: INTRODUCTION

1.1 Overview of the Thesis ........................................ 9  
1.2 Thesis Organisation ........................................ 12

### CHAPTER 2: UNDERSTANDING NOVICE PERFORMANCE

2.1 Introduction ................................................ 15  
2.2 Approaches to Understanding Skill Acquisition .............. 17  
2.3 Production System Models of Skill Acquisition .............. 22  
2.4 Real Beginners in Physics .................................. 28  
2.5 Real Beginners in Programming ................................ 35  
2.6 Other Related Work on Novice Prolog Programmers .......... 40  
  Representing Prolog .......................................... 40  
  Empirical Studies ............................................ 43  
2.7 Conclusions ................................................ 43

### CHAPTER 3: NATURAL AND FORMAL DISCOURSE

3.1 Introduction ................................................ 45  
  Human Discourse .............................................. 46  
3.2 Interpreting Formal Domains ................................ 49  
3.3 Reasoning in Formal Domains ................................ 51  
3.4 Interpretation Processes in Formal Domains ................. 56  
3.5 Mental Models and Natural Language Interpretation ....... 60  
3.6 Logic, Declarative Semantics and Natural Language ....... 68  
3.7 Declarative Programming and Problem Specification ........ 71  
3.8 Mental Models and Problem Interpretation ................. 74  
3.9 'Correspondence' Models and Declarative  
  Representations .............................................. 78  
3.10 Abductive Reasoning ....................................... 83  
3.11 Conclusions ................................................ 86

### CHAPTER 4: A THREE-LEVELLED DISCOURSE FRAMEWORK FOR PROLOG

4.1 Introduction ................................................ 89  
4.2 Levels and Components of Discourse ......................... 92  
4.3 Combining Levels and Components into a Framework ....... 98  
  Input Interpretation .......................................... 100  
  Reasoning Strategies ......................................... 105  
  Output .......................................................... 107  
  Reflective Evaluation ......................................... 108  
4.4 Moving Round the Framework and Representing  
  Misconceptions ............................................... 111  
4.5 Superbugs and Bugs ......................................... 125  
4.6 The Prolog Learner ......................................... 129  
4.7 Learning Prolog by a Declarative Approach ................. 130  
  Input Interpretation .......................................... 131  
  Reasoning Strategies ......................................... 132  
  Output .......................................................... 133  
  Reflective Evaluation ......................................... 134  
4.8 Learning Prolog by a Procedural Approach ................. 134  
  Input Interpretation .......................................... 136  
  Reasoning Strategies ......................................... 136
<table>
<thead>
<tr>
<th>7.3 Section 2: Meta-level Reasoning Superstrategy</th>
<th>229</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Identity Superbug</td>
<td>233</td>
</tr>
<tr>
<td>Wishful Thinking Superbug</td>
<td>240</td>
</tr>
<tr>
<td>Left-to-Right Bias Superbug</td>
<td>244</td>
</tr>
<tr>
<td>7.4 Discussion</td>
<td>250</td>
</tr>
<tr>
<td>7.5 Conclusions</td>
<td>252</td>
</tr>
</tbody>
</table>

**CHAPTER 8: ABDUCTIVE REASONING**

| 8.1 Introduction                             | 255 |
| 8.2 Abduction and the Discourse Framework    | 256 |
| 8.3 Julie and Mary: A Protocol Study          | 257 |
| 8.4 Comments                                 | 269 |
| 8.5 Discussion                               | 270 |

**CHAPTER 9: A CASE STUDY: ALEX**

| 9.1 Introduction                             | 272 |
| 9.2 A Comment on Procedures                  | 275 |
| 9.3 Background Remarks                       | 276 |
| 9.4 Abstract Variables Bug                   | 279 |
| 9.5 Superstrategies and Superbugs            | 284 |
| Conventional Variables Superbug              | 285 |
| 'Lists are brute force' Superbug             | 289 |
| Data and Procedures Superbug                 | 291 |
| 'Member' as a Procedure                      | 299 |
| Where is the Real Prolog?                    | 302 |
| 9.6 Discussion                               | 305 |
| 9.7 Conclusions                              | 306 |

**CHAPTER 10: CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK**

| 10.1 Review of Thesis                        | 307 |
| 10.2 General Comments on Intuitive Strategies in Formal Domains | 308 |
| 10.3 The Discourse Framework                 | 314 |
| 10.4 Superstrategies and Superbugs           | 317 |
| 10.5 Future Work                             | 319 |
| Repeating Studies and Refining Test Materials | 320 |
| Context Dependency and Backtracking          | 321 |
| Visual Programming for Prolog                | 323 |
| Models for Unification                       | 327 |
| 10.6 Afterword                               | 329 |

**APPENDIX 1: Texts of Problem Statements**

**APPENDIX 2: Course Materials**

**APPENDIX 3: Protocol Transcripts for Computers and Thought Students**

**APPENDIX 4: Protocol Transcript for Julie and Mary**

**APPENDIX 5: Protocol Transcripts for MSc Students**
LIST OF FIGURES

Figure 3.1 The Himalayan Tea Ceremony and the Towers of Hanoi.......................... 76
Figure 3.2 Correspondence model of the Towers of Hanoi...... 79
Figure 3.3 The River Problem........................................... 79
Figure 3.4 Correspondence model of the River Problem....... 79
Figure 3.5 Declarative program for the River Problem........ 80

Figure 4.1 Discourses for Prolog........................................... 93
Figure 4.2 Components of Discourse Understanding.............. 96
Figure 4.3 Four Components of Discourse............................ 97
Figure 4.4 The Structure of the Discourse Framework............ 98
Figure 4.5 Input to the Discourse Framework...................... 102
Figure 4.6 Reasoning Strategies in the Discourse Framework... 106
Figure 4.7 Output in the Discourse Framework.................... 107
Figure 4.8 Reflective Evaluation in the Discourses.............. 110
Figure 4.9 A Correspondence Model of a Move in
Missionaries and Cannibals............................................. 112
Figure 4.10 An Alternative Representation of a Move in
Missionaries and Cannibals............................................. 112
Figure 4.11 Ideal movement patterns round the framework...... 114
Figure 4.12 Performance in Deductive Reasoning................ 115
Figure 4.13 Pathway used by Miller's Subjects.................... 117
Figure 4.14 Expected pathway for Miller's experiment......... 118
Figure 4.15 An ideal pathway for formal problem solving...... 121
Figure 4.16 Pathway indicating a 'slip'.............................. 122
Figure 4.17 Pathway indicating more serious error................ 123
Figure 4.18 An Impossible Pathway...................................... 124

Figure 5.1 Pathway for Computers and Thought Students........ 159
Figure 5.2 Computers and Thought Representation Task
Part 1: Prolog to English.............................................. 160
Figure 5.3 Computers and Thought Representation Task
Part 2: English to Prolog.............................................. 161
Figure 5.4 Computers and Thought Representation Task
Part 3: Buggy Clauses.................................................. 162
Figure 5.5 Pathway for Abduction in Interpreting
Prolog Clauses........................................................... 165
Figure 5.6 Coombs and Stell's (1985) Backtracking Example.. 167
Figure 5.7 Modified Coombs and Stell program.................... 169
Figure 5.8 Single Discourse Analysis of Coombs
and Stell task............................................................ 170
Figure 5.9 Dual Discourse Analysis of Coombs
and Stell task............................................................ 171
Figure 5.10 Summary of Observational Studies.................... 173

Figure 6.1 Natural Language Superstrategy and Superbugs..... 177
Figure 6.2 Pathway for Parts 1 and 3 of
Representation Task...................................................... 179
Figure 6.3 Ideal Pathway for Part 2 of the
Representation Task...................................................... 181
Figure 6.4 Actual Pathway Observed for Part 2 of the
Representation Task...................................................... 181
Figure 6.5 Computers and Thought Representation Task
  Part 2: English to Prolog......................182
Figure 6.6 Subject 3's Pathway for Clause Construction........188
Figure 6.7 Pathway for Causal Reasoning in the
  Representation Task..........................195
Figure 6.8 Summary Table of Answers to Last 5 Questions....203

Figure 7.1 Bugs in Mechanistic Discourse for MSc Students...206
Figure 7.2 Meta-level Reasoning Superstrategy
  and Superbugs..................................207
Figure 7.3 Single Discourse Analysis of Coombs
  and Stell task................................209
Figure 7.4 Coombs and Stell's (1985) Backtracking Program...209
Figure 7.5 Results of backtracking task......................210
Figure 7.6 Modified backtracking task......................221
Figure 7.7 Results of modified backtracking task..........222
Figure 7.8 Ideal Pathway for Coombs and Stell Test.........230
Figure 7.9 Meta-level Reasoning Superstrategy
  and Superbugs..................................233
Figure 7.10 Subject 6's Initial Pathway in the Coombs
  and Stell Task.................................239

Figure 8.1 Interpretation Task for Computers and Thought....255
Figure 8.2 Pathway for Abduction in Interpreting
  Prolog Clauses................................257

Figure 9.1 Alex's Pathway Showing Intuitive Moves from
  Logical Discourse to Mechanistic Discourse............273
Figure 9.2 Bug and Superbugs for Alex......................274
Figure 9.3 The Artists Program Specification................285
Figure 9.4 Alex's Artists Program........................296
Figure 9.5 A Snapshot of the Dynamic Tracer During
  Execution of the Append Clauses....................303
Figure 9.6 A Snapshot of the Dynamic Tracer as it Exits
  the Execution of the Append Clauses................303

Figure 10.1 The Three-Levelled Discourse Framework........315
Figure 10.2 Superstrategies and Superbugs Identified
  in the Study..................................318
Figure 10.3 Bugs Identified in the Study....................319
CHAPTER 1

Introduction

1.1 Overview of the Thesis

The main aim of this thesis is to examine the difficulties that beginners have learning the programming language Prolog (Colmerauer, Kanoui, Pasero and Roussel, 1973). It has been claimed (e.g. Kowalski, 1979) that logic programming languages like Prolog are easier to learn and use than other languages because predicate logic is more readily understood than other programming formalisms. Observations of students learning Prolog, however, indicated that many of them did not find it easy.

Present theoretical perspectives from which to view novices and the task of learning to program do not help us to understand the range and complexity of errors that Prolog beginners exhibit. There are two contributing factors to this state of affairs. First, theories of the acquisition of complex cognitive skills, such as programming (e.g. Anderson, 1982), have necessarily tended to ignore some crucial components of the learning environment which affect performance. Such factors include the learner's previous background, the ways in which the learner construes the learning situation, and the role that the body of knowledge to be learnt itself plays in this growing perception. However, for the practical purposes of understanding the often complex misconceptions that our Prolog learners had or developed, such factors clearly did need to be taken into account.

Second, Prolog is closely related to first order logic, and
programs can be viewed either as logical specifications (i.e. interpreted according to the rules of logic) or as a computer program (i.e. interpreted according to the execution strategy of the machine). The logical interpretation requires a high degree of conceptual knowledge before students can fully grasp its relevance to the task in hand, and people seem to have with Prolog similar kinds of conceptual and expressional difficulties as they do with logic. The executional view, although perhaps conceptually slightly easier, is complicated because of the activities of Prolog's automatic mechanisms which the novice may find difficult to control.

In other words, the apparent simplicity of surface features of the language belies both the formal, logical status of the language, and its underlying executional complexity. This means that students tend to write programs which they do not understand because they do not have sufficient understanding at either level. In fact, errors often seemed attributable to combinations of these two factors, making it hard to track down the precise source of bugs in programs.

In order to disentangle all the complexities in this situation a broader view of novices engaging in the task of learning programming is required. The main assumption is that learners do not come to the learning situation as 'empty vessels' even if the domain is new to them. They have powerful intuitive strategies for making sense of unfamiliar concepts and these will be applied to the learning situation. The broader view, therefore, needs to take into account the nature of these intuitive strategies and the effect they have on the beginner's interpretation of the new
Pea (1986) describes 'superbugs' - powerful, language independent, conceptual bugs - which affect the ways in which novices perceive the computing domain. The particular superbug identified in his study is the use by novices of an analogy between human discourse and discourse with computers. This analogy is often unconsciously brought into the learning situation and its effects on learning are far-reaching. In the work reported in this thesis it is shown that superbugs are one facet of 'super-strategies' - high level, domain independent, intuitive strategies which, in their benign form, enable students to make some sort of sense of an unfamiliar domain, but which can turn into superbugs when misapplied.

The thesis has two main objectives. The first is to provide much-needed empirical data on novice programmers learning Prolog. This data relates not only to those novices being taught the declarative semantics, but also those learning via the procedural approach. The second is to develop a framework within which to analyse novice programmer performance building on Pea's notion of 'superbugs'. The framework itself is dependent upon a better understanding of intuitive strategies that novices employ in the interpretation of the learning situation. Because programming is a formal domain, as is logic, the literature on deductive reasoning is used to define a set of characteristic strategies which beginners use to interpret formal domains. The framework relates the various domains of discourse which relate to the activity of programming, each domain having associated with it constraints over actions and interpretations of expressions within the domain.

The question which is then addressed is: to what extent does this
approach explain, elucidate or make coherent, the wide range of errors observed in the novice Prolog programmers participating in this study? It will be demonstrated that a fair proportion of their misapprehensions can be shown to be consistent with the interpretation generated from the discourse framework. This indicates that a major source of misconception is the novice's tendency to interpret both the formal domains of logic, and of programming, using real-world concepts and reasoning strategies. Furthermore, the framework analysis helps to refine and make more concrete Pea's concept of 'superbug'.

This view of the task of programming emphasises the fact that students often lack high-level knowledge about the process of programming, what its intentions are, what its status with regard to formal problem solving is, and, therefore, what kinds of strategies and methods are open to the programmer. Students often do not understand how the various domains in the framework relate, and how to map from one to the other in a way which is relevant to the programming task. The breakdown in mapping also accounts for more concrete errors of interpretation associated with the lower levels.

1.2 Thesis Organisation

Justification for adopting such a view of programming will be offered in Chapter 2 where the disadvantages of narrower views of the task of learning to program are pointed out. This literature will be compared with alternative views of the novice programmer which allow due consideration of prior knowledge and intuitive strategies in learning. The multi-levelled framework is described,
and the concept of superbug is defined. Other research in the area of Prolog novices is briefly presented.

Chapter 3 is a discussion of the literature on deductive reasoning, and it is argued that there is commonality between the errors identified there, and those exhibited by Prolog learners. This commonality derives from the characteristic use of natural language semantics to interpret formal domains, and the tendency to confirm rather than disconfirm hypotheses. The ramifications of this claim are explored in relation to declarative interpretations of Prolog, the declarative method of problem specification, and abductive reasoning.

In Chapter 4 the framework is introduced and developed. The main conclusion is that human discourse processes can, and often do, interfere with learning a formal skill such as programming, or logic. In the programming domain this has two major effects: the first is to confound the status of the language, and the second is to misinterpret the behaviour of the computer, and to treat it as an 'other' in a discourse process which is more akin to human/human discourse than human/computer discourse. This chapter concludes with the hypothesis that the two main methods of teaching Prolog - the declarative and the procedural - may contribute to both types of misperception. In the case of declarative approaches to Prolog, students are likely to misunderstand the semantics of expressions in the language. This will lead to incomplete programs. In the procedural approach, students are less likely to fall into the language difficulties, but their ability to correctly predict the machine's behaviour as a program is run is affected by their tacit attribution of intentionality to it.
Chapter 5 presents the methodology used for observation of students learning Prolog.

Chapter 6 presents data obtained from undergraduate students who were taught declaratively. It is shown that their errors are attributable to misconceptions about the language of programming discourse. That is, they interpret programming expressions as if they were natural language, and create programs as if they were to be interpreted by another human, rather than a machine. This leads to incompleteness of specification.

Chapter 7 reports on data from post-graduate students who were taught Prolog procedurally. These students are shown to have several interfering factors from general discourse in their prediction of the computer's execution strategy. The data from this study is analysed first from the perspective of programming discourse i.e. where the emphasis is upon concrete bugs in novices' interpretations of a Prolog program. The same protocols are then analysed from a higher level view to demonstrate the presence of, and effect of, superstrategies and superbugs.

Chapter 8 presents a protocol which illustrates the effect of abductive reasoning on program interpretation in two undergraduate students.

Chapter 9 is a case-study of a post-graduate student who had previously learnt other programming languages before learning Prolog.

Chapter 10 presents conclusions and recommendations for future work.
CHAPTER 2

Understanding Novice Performance

2.1 Introduction

Observations of Prolog novices shows that they experience an extremely wide range of problems, not all of which are amenable to analysis by traditional methods. This suggests that learners are engaging their prior knowledge, experience and general purpose problem solving strategies in the early stages of learning. But it seems also that those students who are apparently doing well enough sometimes turn out to have misconceptions which eventually emerge to cause problems at a time when the student least expects it. So although the early phase of learning is our major focus of interest, another concern is how misconceptions laid down at this time affect later performance.

Theories of skill acquisition are discussed in this chapter, and aspects of the learning situation which are not taken into account by such theories are pointed out. The major factors which are ignored are: 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 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.

In this chapter, consideration is given to some classical approaches to the study of novices learning to program, and these are contrasted with investigations of novice performance in other formal domains. The major points are:

1: studies of programming skill acquisition have tended to ignore external influences on the novice's perception of the task;

2: due to the lack of appropriate domain-specific knowledge, novices are obliged to use intuitive reasoning strategies in formal domains, but these strategies are not random;

3: intuitive strategies are often powerful but can also sometimes lead to misconceptions or errors which are described as 'superbugs'.

First, it is important to clarify what is meant in the discussion by the term 'formal domain'. Larkin (1981) defines formal domains as follows:

...domains involving a considerable amount of rich semantic knowledge but characterised by a set of principles logically sufficient to solve problems in the domain. (p.311)

In other words, the learner is confronted with a body of knowledge which has logical coherence but which may not contain explicit
information regarding how and when to use constructs appropriately. Learners must infer this information for themselves. These issues have been explored most extensively in the domain of physics problem solving, and this literature will be referred to as appropriate. Similarly, the study of deductive reasoning in non-logicians offers some insights into commonly occurring mistaken assumptions and errors in performance.

The students of particular interest are those who have no special formal training or strong mathematics/science background. di Sessa (1977) very aptly describes the problems for beginners learning how to use formal systems. His major points are: (i) teachers of such systems often believe them to be very simple and therefore often respond inappropriately to beginners' problems; (ii) beginners need to be explicitly orientated toward understanding the framework; (iii) the meaning of expressions in the domain may not be at all obvious to the beginner; (iv) learners may eventually become alienated and disillusioned by the apparent arbitrariness of such systems. The following discussion is addressed mainly to point (ii) by suggesting that learners do not understand what is 'appropriate' within the formal domain in which they are working - in other words, their general problem solving skills are not 'tuned' to working with formal domains.

2.2 Approaches to Understanding Skill Acquisition

The overall criticism of previous research in skill acquisition is that the methodological framework in which it is conducted precludes analysis of the very early but crucial phase of learning involved with reading, understanding, and appropriately
interpreting the problem to be solved. Studies tend to focus too narrowly on fine details of cognition occurring in a very limited phase of the learning process - i.e. after the learner has understood what to do. That is, the work is based on the assumption that the novice is orientated towards working within a specific domain and is not trying to cope with other extraneous interfering factors.

Experimental work in the field has mainly taken place within the cognitivist paradigm with its emphasis on the structure and limitations of human memory as being the biggest constraint on performance in cognitive tasks. Many models of the learner are based on Newell and Simon's (1972) theory of human problem solving where memory is conceptualised as being divided into:

* short term memory (SIM) - a buffer with limited capacity and rapid decay

* a theoretically unlimited, long-term memory (LIM) - which requires rehearsal to record input

* working memory (WM) - where information from SIM and LIM is combined to produce problem solving behaviour

Some researchers accept Miller's (1956) thesis that the limit on the capacity of SIM is around seven, plus or minus two, 'bits' or 'chunks' of information.

For example, Shneiderman and Mayer (1979) present a cognitive model for describing expert programmer behaviour which is intended to address the issues of: problem understanding, program composition, debugging, program modification, the cognitive structures that programmers possess, and the cognitive processes involved in using this knowledge or in adding to it.
They hypothesise the presence of 'multilevelled, funnelled cognitive processes' where the expert programmer has in LTM semantic and syntactic components: semantic memory contains high-level, language-independent planning information, whereas syntactic memory contains language-dependent information about specific details of programming languages known to the individual. They do not discuss how knowledge entered the cognitive system in the first place, although they maintain that semantic knowledge is learnt 'meaningfully' whilst syntactic knowledge is 'rote-learnt'. The first step in program composition is to get the relevant information into working memory from SIM and LTM - how this is achieved is not described. The programmer then devises a plan, first in general programming strategies (from semantic memory), which is then 'funnelled' into specific generation of code (syntactic memory). The general plan is referred to as 'internal semantics'. Internal semantics, once established, are resistant to forgetting and can be transformed in a variety of ways. Thus, the coding of a problem into any particular language becomes a trivial matter.

This view has been criticised by Arblaster (1982) who points out that the semantics of a particular language can have a great effect on the sort of problem solution at which one can arrive.

At another level of criticism this model may be descriptive of certain facets of expert performance but it does not satisfactorily explain the mechanisms by which expertise is attained. Information from LTM appears to enter SIM in this model which seems psychologically implausible if SIM is a buffer with limited capacity and rapid decay (Newell and Simon, 1972) because in order
to keep such items in SIM, constant rehearsal would be needed, thus using up valuable 'processing power'. Furthermore, working memory takes on almost the entire role of cognition in ways reminiscent of the 'homunculus' which has haunted cognitive psychology for decades. In working memory the problem is 'analysed and represented' in terms of given state and goal state whilst at the same time 'general information from the programmer's long-term memory (both syntactic and semantic) is called and transferred to working memory for further analysis'. Whilst this description may be appropriate in terms of a production system it does not explain in the human case how information is 'selected', or what processes do the 'analysis'. No account is given of the effect of perceptual processes entering SIM and therefore no factors in the environment can be seen to impact upon performance. The system relies on some form of 'pre-processing' which is not accounted for or located within the system.

Shneiderman and Mayer's paper is fairly typical in the field of HMI and illustrates some of the main deficiencies of research in this area. Based on box-model diagrams it points to the various places that 'processing' can take place, the entry and exit ports of major cognitive events, and explains errors in terms of the diagrams (c.f. Neisser, 1976, for criticisms of box-model theories). The basic cognitive framework, whilst it is valuable in the study of certain psychological phenomena, is too impoverished to cope with the complexity of a task such as learning to program. Sheil's (1981) article criticises experimental work in this field and its lack of reliability and generalizability for precisely this reason as well as for methodological infelicities.
In fact, the stronger criticism has been made that the cognitivist methodology, because of its emphasis on memory structures, has had a detrimental effect on understanding programmers because it has narrowed our view of what it is to learn (Rumelhart and Norman, 1981). Although memory structures are obviously important in learning, modal models of memory (e.g. Atkinson and Shiffrin, 1968; Baddeley, 1976) are too simple to be useful in the enterprise of understanding cognitive skill acquisition. For example, a major aspect of learning is the integration of new knowledge with existing knowledge but the cognitivist framework as used in the study of programming does not allow for consideration of the processes involved in this. The learner arrives as a processing system with no previous history or experience. The various systems presented which represent the process of skill acquisition contain no other kinds of knowledge to conflict with incoming information except in terms of local problems associated with syntactic differences between particular languages. Knowledge integration is probably the most important aspect of learning a programming language since the internal logic of particular constructs is often not difficult. What is hard is appreciating how to use such a construct which depends on understanding how it relates to other knowledge - either domain dependent programming knowledge or domain independent general knowledge.

Experimental hypotheses generated from this framework are therefore premature given the current state of our understanding of programmers and the task of learning to program. Too many fundamental questions are left unanswered. It is crucial to understand a great deal more about the high-level strategies that
beginners bring with them and the kinds of intuitions they apply when trying to understand a new domain.

2.3 Production System Models of Skill Acquisition

A more complete exposition of skill acquisition is that of Anderson (1982), whose ACT* theory has enabled him to model certain facets of the learning process using production system methodology (GRAPES). Anderson's theory directly addresses both the process of transforming declarative information (for example from textbooks) into a procedural form, and the mechanisms which enable the initial approximation to be transformed into action patterns, thereby producing rapid domain-appropriate behaviour. Because of the emphasis on mechanisms it represents a great advance on box model theories and can offer a more detailed account of the processes involved.

Anderson proposes that skill acquisition has two major stages: 'a declarative stage in which facts about the domain are interpreted and a procedural stage in which the domain knowledge is directly embodied in procedures for performing the skill'. The transition from declarative to procedural is achieved by 'knowledge compilation'. The theory assumes that the human cognitive system has general interpretative procedures with no domain-specific knowledge which can be applied to facts (declarative representations) to produce coherent and domain-appropriate behaviour. All new knowledge enters the system in a declarative form, is stored declaratively in memory, and is interpreted by these structures. Declarative information does not have any direct impact upon behaviour except when interpreted by the structures.
Initially, the learner 'approximates' the skill by hand-working step-by-step through given instructions. This process is slow because of the need to refer to instructions and the associated risk of losing previously computed information from short-term memory in the process. However, the 'interpretative procedures' impose flow of control on the declarative instructions and result in a 'proceduralised' version. Having approximated the skill the learner chunks information and then begins to collapse sequences of actions which commonly occur close to one another treating them as a single unit of behaviour and no longer referring to instructions. This is the process of knowledge compilation.

Errors can occur when the student compiles action sequences which may occur together but which are in fact unrelated. Anderson believes that holding declarative instructions in memory for a while for reference purposes gives the learner an opportunity to test out possible compilations before becoming committed to them. Once the skill is compiled the student rarely needs to refer to the declarative representation in memory except when forced to by meeting a new situation, or when anomalous behaviour is produced by the machine. As expertise matures the declarative representation is thought to fade altogether giving expert performance an appearance of 'planlessness'. Anderson accounts for experts' inability to verbalise their knowledge by suggesting that compiled procedures are inaccessible to conscious inspection. Adelson (1984) describes this process concisely:

...information that is well known to the subject can be represented as a procedure that can be used to match, and thereby recognize, incoming information very rapidly. However, knowledge contained in a procedure cannot be inspected directly; what the knowledge is can only be
inferred by noting what the procedure does. Having developed these procedures, the information comes to be represented in a way that hides the details of the processing to be done.

Compiled proceduralised knowledge undergoes continued refinement which results in increased speed of execution. This additional learning, called 'tuning', essentially serves to reduce the search space for solutions.

Anderson has illuminated many aspects of the process of establishing the basis of skill through proceduralisation and compilation. However, related assumptions made as part of the theoretical framework limit its application to understanding 'real' beginners. These assumptions are: (i) that there are existing interpretative procedures which are applied to new information and provide a flow of control; and (ii) that 'hierarchical control of behaviour derives from the structure of problem solving' and that 'problem solving and the goal structure it produces is a fundamental category of cognition'.

The first assumption Anderson justifies by reference to observations of students performing a task in geometry. He noted that when presented with instructions for generating proofs in geometry, non-mathematics students developed a procedure and performed the task, slowly, but with a clear flow of control, where no control information was explicitly stated in the instructions. This, he suggests, is achieved through the use of general problem solving methods (such as searching lists of methods) and combining declarative information with the set of instructions to produce novel actions.

Although Anderson stresses that these general procedures are not
the only ones used in learning he does not examine their nature, what informs them, and ways in which they can lead the learner astray. There are very many factors which influence the kind of interpretations students put upon a task and some of these influences will be discussed in more detail in Chapter 3.

But, as an example, Anderson claims that computing languages like LISP do not have any analogies outside of programming. This may be true in a general sense but it does not stop beginners from introducing analogies which they think are appropriate because of the interpretation they have made of the learning situation. For Anderson, analogical learning processes are brought into play by providing the student with templates of simple LISP functions, and the claim is that students need only work through a single example to understand a class of examples. This is described as 'one-shot' learning. This finding, which has not been replicated elsewhere, implies that the kind of subjects that Anderson studies are ones who are already 'semi-skilled' because our observation of learners suggests that they tend to force all kinds of analogies into learning programming, appropriate or otherwise. Students who can confine their analogical reasoning to LISP examples alone are already well on the way to being semi-skilled and are not representative of the learners we have observed.

But the control flow derived from interpretative procedures is itself dependent on problem reduction by 'sub-goaling'. The ACT* system is based upon a top-down goal/subgoal structure of sub­routines. Systems such as these have enabled us to model many aspects of cognition. But caution should be exercised in assuming that such systems are actually providing us with an adequate
account of cognitive processes, for the simple reason that although problem reduction seems often to be employed by students, it is only one strategy amongst many, and it can easily be disrupted by external factors which should be taken into account.

For example, goal/subgoal structuring works well only if the information within the system is relevant. Unfortunately, human problem solvers tend to introduce into the problem-solving domain anything and everything they think will assist them in achieving a solution. In this case, goal/subgoal structuring of behaviour becomes a very random affair because students appear to draw information from a variety of sources when deciding what to do next, and clear hierarchical control flow is not evident in their performance. Anderson's system succeeds because it has not only all the information it needs, but only the 'correct' information. It does not thrash about trying to decide what the problem 'means'.

To illustrate this point further, using a similar methodology to Anderson, Larkin (1981) demonstrates how general 'GPS'-like (Newell and Simon 1972) means-end analysis problem solving methods can combine with domain knowledge of the sort to be found in textbooks, to produce selective, domain-appropriate behaviour. She notes that students of algebra and physics use general purpose problem solving strategies applied to the specific principle given as part of the problem statement. She describes a simple learning mechanism which allows a production system (ABLE) to enrich its formal knowledge of physical principles through practice. Initially the production system has knowledge of principles, knowledge of how to interpret the symbols in the principles, and an algebraic means-end analysis method of problem solving. In solving equations,
the ABLE system must search its entire knowledge base for principles related to the given problem proceeding from known quantities in the givens to the unknown quantities required. As it practises on examples of problems the ABLE system builds new production rules that match the specific situation in which various principles apply. These specific productions are more powerful than the general ones provided and they can be applied in appropriate situations very rapidly. Eventually, therefore, instead of general and individual principles the system creates 'bundles' of knowledge about the application of particular principles in specific circumstances.

Although this system again illustrates some important facets of novice performance in physics problem solving, it starts with a 'correct' (though admittedly rather primitive) initial representation, and contains only relevant information. The theory does not take into account the wide range of influences which can affect performance, such as previous experience, misinterpretation of instructions, the interpolation of inappropriate knowledge and so on.

It is shown below that these theories, whilst being informative with regard to some of the processes involved in skill acquisition, are not able to account for a wide range of possible errors or misconceptions that beginners have because their view of learning-by-doing which consists only of establishing procedures is too narrow. It is essential to understand how other kinds of existing knowledge are brought into play. The processes of proceduralisation and compilation are likely to be relatively low-level mechanisms for establishing control over execution of behaviour, and these
have only been described in the context of learners who are already at a stage where they can appropriately interpret problem descriptions within a particular domain. Anderson's theory addresses a level of learning which must ignore general misconceptions that students may have of the task, and of the inappropriate interpretations that students are shown below to force onto instructions. However, since these high level factors frequently disrupt the process of skill acquisition, it seems essential to include them in an analysis of novices learning to program.

2.4 Real Beginners in Physics

The fact that learners do not come to the learning situation as 'empty vessels' even when the domain is entirely novel to them has been noted by many researchers. People have strategies and techniques for trying to understand new domains; they have previously acquired knowledge and experience with learning situations; and they have beliefs and theories about the world and the way things work. They also do not start the task with only correct and relevant information to hand. Information pertinent to real-world, practical problem solving is often used by novices to interpret the formal domain and can cause students to misperceive the domain in which they are supposed to be working.

In support of this view, Scanlon and O'Shea (1986) argue that deciding upon an appropriate representation for physics problems is a major obstacle for novices. They emphasise that the learner's initial representation of the problem is subject to external influences - for example, assumptions about the task based on
previous experience of school-room activities, and misreading the problem. Real learners have to formulate a representation of the task, a process which can be fraught with errors. Decisions must be made about how best to go about the perceived problem which may not correspond directly to anything that has been explicitly taught or which is contained in text-books.

Furthermore, having got as far as adopting some kind of representation, novices frequently hinder their progress by changing representation mid-stream. In other words, students may have competing representations from which to choose and may not know how to structure their problem solving strategies effectively. Scanlon et al. observed that experts minimise the number of transitions they make from one representation to another, whereas novices tend to behave in what the authors describe as a spendthrift way, changing representation when encountering difficulty rather than backtracking. This sort of behaviour is unlikely to exhibit hierarchical goal/subgoal control flow.

Similarly, Elshout, Jansweijer and Weilinga (1986) criticise Anderson's work for being too low-level an analysis and develop the notion of a 'genuine beginner', calling for the study of domain independent novice problem solving to be a field in its own right:

...there is evidence that what characterises the beginning problem solver has generality...[and] can be explained in general psychological terms not bound to the specific problem domain studied... (p. 1)

They identify the trait of 'wishful thinking' in many beginners - for example, they observed students learning to play billiards, who, when asked to account for the shots they had made gave answers which implied that a ball which was obstructing their path would
conveniently move out of the way, although they must also have been aware that it would not. Similar cases of wishful thinking are also found in protocols of students solving physics problems and programming problems.

Genuine beginners, then, are located in the context of understanding complex, often formal, systems through learning-by-doing with possibly only a text-book or manual to refer to and their own general problem solving skills to deploy in the task of understanding what to do and how to do it.

Elshout et al. assert that introductory texts and manuals are only 'complete' in a formal sense. In other words, although they may contain all the necessary information from an expert's point of view, to the beginner they are 'full of ambiguity and loose ends' because they do not advise how and when to make certain kinds of assumptions to create a theoretical representation of real life systems. They note the 'meandering GPS-like' search of novices and point out that in contrast to Anderson's learners, who are able to interpret instructions and execute a task, many learners trying to solve problems in a new domain show very different behaviour, characterised by random casting about for ideas, guesswork (sometimes informed, often not) and 'wishful' thinking.

But apart from simple lack of appropriate knowledge and lack of 'know how' beginners are also likely to have their own naive conceptions or intuitive 'theories' which are introduced into the problem solving episode as confounding issues. Learners therefore spend most of their time cycling through a reason-and-search process during problem interpretation, a process which for Anderson
is only the initial phase of skill acquisition. Elshout et al. emphasise that all beginners must use general, domain independent GPS-like strategies, but that the protocols of genuine novices are not characterised by careful stepping through of instructions which lead eventually to smooth execution. They further point out that the progress from 'idea to act' in real life domains is not a smooth progression as Anderson describes it, but, rather, is continually interrupted by impasse/repair cycles (Brown and van Lehn, 1980). This is not regarded as detrimental: it is noted that whilst things are going well (or apparently so) learners tend not to take the opportunity to explore concepts they are using. They use them because they have been advised to either implicitly by previous successes or explicitly by teachers and manuals. Elshout et al. suggest that the repair cycle may provide a context in which to explore how to use some knowledge and why it is appropriate to do so.

In order to model this form of novice behaviour, Elshout et al. require a two-levelled system: the first level problem solver is domain specific, the second uses general domain independent problem solving strategies based on repair theory where, upon hitting an impasse, the novice begins the reason-and-search cycle. The meta-level domain independent rule set is specifically tailored to reflect the conceptualisations (and misconceptualisations) of the subject under study, thus catering for the impact of the individual's prior knowledge on performance rather than trying to eradicate it from the experimental situation.

This positive view of the novice's contribution to the learning situation is shared by di Sessa. Rather than fighting the flow of
intuitive interpretation di Sessa (1977) suggests ways in which intuitive understanding can be developed into formal understanding. From the slogan that 'people are more fundamentally model builders than they are formal system builders' di Sessa argues for a procedural approach to teaching formal skills, where existing practical knowledge is deliberately brought in to play:

Developing in the style of and from the contents of heuristic and intuitive world models is a very robust structuring of knowledge. Though personal 'world models' may be mistaken in many details and may seem on the surface quite imprecise, they are secure in that they do not arise from abstraction but from procedures which work. (p.17)

di Sessa (1982) argues that in order to be able to capitalise on existing knowledge it is crucial to understand how that knowledge becomes involved in understanding some formal concept. To achieve this end he distinguishes 'abstract task analysis' from 'genetic task analysis'. Abstract task analysis is used to determine the knowledge content of subject matter whereas genetic task analysis is used to identify components of pre-existing knowledge which can be used in understanding a formally expressed concept. Abstract task analysis, he argues, tends to produce illusions in terms of the organisation or 'newness' of ideas or concepts - in other words, the analysis may produce the 'concept of velocity' as if it were entirely new, whereas, in fact, everybody has a naive understanding of the behaviour of moving objects.

So, whereas abstract task analysis identifies the logical prerequisites for task performance, genetic task analysis identifies 'genetically antecedent, partial understandings' which the teacher can shape through intervention. This approach puts emphasis upon the kinds of 'natural learning pathways' that users
might need to take as they work towards a full grasp of formal concepts expressed in the terms of a formal language.

Thus, di Sessa (1982) speculates that although one may have a 'parsimonious description of a concept' (i.e. the formal expression of it) the way in which that knowledge is used functionally:

...will involve a confluence and complex orchestration of a large number of partial understandings, with many of them based on previous knowledge. We will use the term 'distributed encoding' to emphasize this fragmented view of knowing which we take as important to following the genetic lines of understanding. (p.59)

Examples of classes of knowledge which are encoded in a distributed way, contributing to functional understanding, but which are not part of the formal expression of the theory include: experts' knowledge of special cases which are used so often that they are no longer deduced from the theory; particular ways of interpreting a context to realise that some technique is appropriate; the use of examples and counter-examples.

Extending his analysis to the computing domain di Sessa (1985) argues that models of programming languages should accumulate through existing partial understandings and illustrates the case by reference to LOGO syntax. He claims that LOGO syntax systematically uses English as a knowledge frame for understanding. For example, in the procedure to make a turtle draw a square forever:

```
TO SQUARE :SIDELENGTH
FORWARD :SIDELENGTH
RIGHT 90
SQUARE :SIDELENGTH
END
```

he points out that the line 'TO SQUARE :SIDELENGTH' exploits the fact that English uses infinitives definitionally. The knowledge
frame can be disrupted by changing the syntax, viz:

```
DEFINE "SQUARE [[SIDELENGTH] [FD:SIDELENGTH]
[RT 90] [SQUARE :SIDELENGTH]]
```

This second definition would be used by a more sophisticated programmer for whom the syntax items have meaning not only in terms of the system changes, but also in terms of uniformity with other kinds of syntax statement. Di Sessa emphasises that the knowledge frame of English for LOGO is not likely to be persistent - i.e. it will soon be discarded for a more subtle understanding of syntax items, and recognises the fact that learners may get into trouble using this analogy:

No learner believes LOGO is English for very long. But we must be aware that globally destructive misconceptions may be fostered as well as misconceptions which, ironically, are profitable. (p.13)

This is to emphasise that prior knowledge brought by novices to the programming domain can not only be of great value, but can also lead to difficulties.

The important points in the above studies are that the learner's knowledge state is regarded as an important variable; that formal knowledge is quite different from intuitive practical knowledge; that simple presentation of correct information may not be appropriate for beginners since they may not be able to leapfrog a necessary stage of understanding; and that functional knowledge may involve sets of partial understandings. All of the researchers mentioned above point to the fact that a full understanding of novice performance can only be achieved through the use of multi-levelled, or multi-faceted models.
2.5 Real Beginners in Programming

The real beginner in programming is now considered. An obvious confounding factor for the novice in the programming domain, as opposed to novices in other formal domains such as physics or logic, is that they have an additional burden of coping with a complex interactive device - the computer. Interactive programming involves not only attempting to understand the problem domain and the programming language but also grappling with a machine whose behaviour is often difficult to predict. They have to find ways of conceptualising the machine with which they are working, its various modes, the multiple systems that run on it, and ways in which these all relate to the task.

The machine's behaviour will be a salient factor in the novice programmer's problem solving situation since it provides a source of immediate feedback and, after all, the novice's task is to make this mechanism behave in a certain way. This presents the novice with, as it were, yet another level of problem solving: to interpret the computer's responses in a way consistent with the problem situation as she has defined it. Creating causal explanations for the machine's behaviour plays an important role in learning how to use it (Waern 1984) but, unfortunately, there are often sharp distinctions - which the novice does not perceive - between what she thinks she has told the machine to do and what she has actually written in the program. Many disparities between what the novice thought the computer would do, and what the computer actually did, can be related to misguided expectations that learners have about the behavioural capacity of the machine.
Intuitive strategies are essential for beginners in this complex situation but, as has been pointed out, their use can also lead to difficulties. Pea (1986) describes how students bring their intuitions about human conversational discourse to the programming task:

Specifically, students have a predominant analogy that guides their behaviour when, as novices, they write programming instructions to a computer. This analogy is conversing with a human. Their pragmatic strategies for using natural language with other humans lead them astray as they try to deal with programming, because programming is a formal system that interprets each part of a program...in terms of rules that are mechanistic. (p.26)

Pea points to the way in which formal languages violate conversational maxims and the lack of inferential capabilities on the part of the machine. The main consequence for programmers is what Pea describes as a 'superbug'. A superbug is a persistent language-independent conceptual bug. The superbug he identifies is the default strategy that

....there is a hidden mind somewhere in the programming language that has intelligent interpretative powers. (p.25)

Pea stresses that too facile a reading of this statement should be avoided. It is not that students literally think there is a mind inside the machine. They frequently are aware that there is not. The point is that their behaviour often seems to contradict this knowledge - they act as if the language can do much more than it can.

Three major classes of bug are pinpointed as resulting from this superbug:

* Parallelism bugs
Parallelism bugs are associated with the misconception that several (or all) lines in a program are 'active' or are known by the computer at the same time. This leads to errors in the construction of loops, for example, where lines of code are left outside the loop as demons waiting for some condition to be fulfilled instead of being embedded within the loop. Students think that the computer can remember that it has read such a line and will return to it under the right conditions.

Intentionality bugs show up when students are asked to predict what a program will do and are associated with attributing to the program foresightedness or goal-directedness. So, again, the program is described as in intentional terms: it 'wants' to do things, or it 'has to stop'.

Egocentrism bugs occur during program creation when students assume that their program is as meaningful to the computer as it is to them - i.e. they think that the computer will be able to fill in missing, but obvious, details. The source of all these bugs is the assumption on the part of the learner that the machine can understand much more than it can.

Pea's analysis provides a convenient way of describing the effect of prior knowledge on programming and the basic ideas he presents will be refined and developed in relation to Prolog novices, although some redefinition of terms will also take place.

When applying this kind of analysis to Prolog learners, though, it becomes apparent that their situation is somewhat complicated by
the fact that the language can, to some extent, make inferences, and that it is goal-directed. For example, Pea points out that the parallelism bug is liable to be differently manifested in Prolog than in other languages because the sorts of expectation which it generates are frequently met in Prolog. His example in Pascal to illustrate parallelism at work is as follows:

\[
\text{AREA} = \text{Height} \times \text{Width} \\
\text{Input Height} \\
\text{Input Width} \\
\text{PRINT } "\text{AREA}\"
\]

where the student assumes that the program will use the values of Height and Width which arrive as input as the values in the multiplication. However, the values for Height and Width will remain zero since they have not been correctly initialised. But, as Pea comments, this program would translate easily into a working Prolog program:

\[
\text{Program: area}(\text{Area}, \text{Height}, \text{Width}) :\text{- Area is Height} \times \text{Width}. \\
\text{Query: } ?- \text{area}(\text{Area}, 8, 12).
\]

In this case, the value of the input variables Height and Width are matched into the body of the clause, the multiplication is performed, and the variable Area is instantiated (by use of the in-built operator 'is') to the result. Since the variable Area is acting the role of an output variable in the head of the clause, the result will appear on the terminal screen:

\[
\text{Answer: } \quad \text{Area} = 96 \ ?
\]

Even those unfamiliar with Prolog would probably agree that the above clause looks reasonably straightforward to read. But, unfortunately, this does not mean that Prolog novices are liberated from the parallelism bug. Creating this apparently simple clause
and understanding why it works requires a fair amount of knowledge. It does look, for example, as if the statement on the right is a repeat of the statement on the left. Why could it not have been written:

\[
\text{area(Area, Height * Width)}.
\]

The answer lies in realising that, in the correct version, the left hand side sets up the variables, whereas the right hand side does the computation. As it stands, the above clause is simply a Prolog term which can match objects but which does not evaluate anything. There is, therefore, no result to be returned.

So although the parallelism bug may not be manifested in quite the same way as it is in other languages, it may be that situations where its expectations are met are still problematic. That is, students may never really stop to think how an effect was achieved because at the time of being shown it was thought to be obvious. It is only when they come to create those clauses themselves that they realise how much knowledge they lack. At this point, they may 'reason and search' in the manner described by Elshout et al. and thereby introduce all kinds of misapprehensions. This issue will be discussed further in later chapters.

Pea's main points are (a) that learners do have implicit skills which they bring to the learning situation but these sometimes are inappropriate to the task in hand, and (b) that comparatively low level bugs are often symptomatic of higher level misconceptions. The behaviour of the beginner viewed as a whole can show major sources of misconception which all too frequently are attributed to low level errors but which are, in fact, due to very high level misconceptions resulting from the misuse or misapplication of high
level strategies. In other words, superbugs cause bugs: fixing the bugs is one way of dealing with this situation, but it would be preferable to address the issue head on by investigating the nature of superstrategies. In this way we can then identify situations where superbugs may arise.

It is this broad approach to analysing novice Prolog programmers which is adopted in this thesis. It is similar to that of Elshout et al. (op.cit.), and Kahney (1982) who argues that models of novices in domains such as physics, derived from original work on chess experts (Newell and Simon, 1972; Chase and Simon 1973), have yielded much more coherent pictures of their behaviour, knowledge and strategies than have typical analyses of programming novices. He suggests that current approaches to understanding novice programmers force artificial distinctions between experts and novices, since many of the tasks provided in experimental situations tend to favour experts, making novices appear as 'know-nothings'.

2.6 Other Related Work on Novice Prolog Programmers

There are several other research projects in progress investigating the behaviour of novice Prolog programmers, and ways in which they can be supported.

Representing Prolog

Bundy, Pain, Brna and Lynch (1986) are trying to develop 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. They indicate that the difficulty in finding a suitable representation is that there is so much going on during execution that it becomes difficult to display by conventional means - i.e. by use of blackboards or acetate slide presentation - and normal terminal screens are often too small to display sufficient information at any one time about the whole execution process to be of any great use. The conventional debugging trace which accompanies most Prolog systems is also difficult for beginners to use because the trace output is rather uniform, dense and detailed. They conclude that AND/OR trees are the most useful representation for students, used in conjunction with a view of the database, and a resolution table which assists in keeping track of variable instantiations.

Eisenstadt and Brayshaw (1986) 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.

Building automatic debuggers and intelligent tutoring systems for Prolog is a difficult task. As Ross (1986) points out, it will be hard for a debugger to determine whether input is syntactically complete or not, and, unless the intentions of the programmer are known, cases will arise where it is just not obvious where the error lies. Research is still in progress on this issue.

Rajan (1985), 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 dynamic trace. We can certainly confirm that the single stepping dynamic tracer incorporated into the POPLOG system proved invaluable in helping students understand execution, particularly with the classic 'difficult' definitions of 'member' and 'append'. The advantage is that the tutor can run the tracer focusing the student's attention on a certain aspect of the process - e.g. variable instantiation - whilst at the same time it is obvious that a great many other activities are taking place. Subsequent runs through can allow the student to focus on other facets of the execution. The sometimes frantic activities of the tracer can be a healthy reminder to the student that processes such as instantiation do not take place in a vacuum. Unfortunately, though, the tracer can really effectively only be used on very small programs since the tree rapidly goes off the screen on larger programs.
These studies of Prolog programmers are geared towards teaching issues and most studies (except for Rajan's) are non-empirical.

**Empirical Studies**

Other empirical approaches in this area are those by Van Someren (1984; 1985) and Ormerod, Manktelow, Steward and Robson (1984). Van Someren is investigating the 'mal-rules' novice programmers have. In many respects his work is complementary to that reported here, providing a perspective of Prolog viewed from the lower domain levels.

Ormerod et al. are investigating the effect of list representation in Prolog on reasoning skills.

Coombs and Stell (1985) have been investigating misconceptions of novice programmers with a view to building debugging tools. Studies they have made of backtracking errors are used as a basis for protocol studies in Chapter 6 of the thesis.

**2.7 Conclusions**

It has been shown that existing theories of cognitive skill acquisition cannot help us understand the range of difficulties experienced by novice Prolog programmers. In many respects, skill analysis focuses on the successful student who is able to understand the purpose of the enterprise and who has only domain specific difficulties to obstruct learning. It has been pointed out that those researchers who are interested in real beginners have to address existing knowledge and high-level strategies which beginners bring to the learning situation which may prevent them from ever reaching the stage where they can develop the skill. This
requires a multi-levelled view of novices and the task in which they are engaged.

Beginners cannot be expected to throw off all their previous experience and knowledge when they enter the learning environment. It is essential to understand what kinds of common assumptions beginners make and how these help or hinder their progress.

Programming languages are communicative systems however impoverished they may seem in relation to natural language. The problem-solving cycle, therefore, is to perceive and correctly interpret the required sequence of actions in a given problem, to specify those actions in such an explicit manner that the computer can execute them, and then evaluate the resulting behaviour for its appropriateness. The degree of 'appropriateness' will vary according to the success with which the first part of the cycle has been carried out.

It is not surprising, therefore, that students use their very deep knowledge of human discourse in the programming domain since there are very few other situations which require this amount of sophisticated interpretation. If such knowledge can make the programming task easier, a genetic task analysis would help to establish the sorts of partial understandings which can be usefully employed. A precursor to this, however, is to analyse what beginners presently do and conditions under which their intuitions help or hinder their progress. This general approach seems more appropriate to a fuller understanding of novice performance than the limited view of novice programmers and the learning process to be found in more traditional theoretical approaches.
CHAPTER 3

Natural and Formal Discourse

3.1 Introduction

In the previous chapter it was argued that conventional analyses of novice programmers tend to ignore the strategies and knowledge that beginners bring with them to the learning situation. The most interesting questions about novices learning are therefore left largely unaddressed: in the absence of correct and specific domain knowledge, how do beginners construe statements in programming languages, and what do they think those statements mean to the computer? In other words, how do they interpret the programming situation, and in what way (if any) does this affect their learning?

Pea (1986) describes novices using the analogy of human-human discourse to understand human-computer discourse. His claim is that novice programmers work 'intuitively' and his paper constitutes a first step towards explicating what it means to work intuitively. He describes how programming languages 'systematically violate' human conversational maxims because, for example, a computer cannot 'infer what a speaker means if she is not absolutely explicit, whereas a listener in human-human conversation can query the speaker for clarification' (p. 26). (It should be pointed out perhaps that computers do give error messages which could be interpreted as an abrupt way of demanding clarification!)

This idea, itself an analogy - it is as if students use an analogy between human discourse and programming discourse - is an extremely
useful one, but there are aspects of it which still need drawing out. As it stands, we can do little more than bemoan the fact that this tendency exists. In order to push the analogy further, and in the spirit of di Sessa's work with physics novices, it would seem useful to begin constructing a genetic task analysis of programming where the starting point is to accept that novices do use human discourse interpretation strategies in the programming domain as a superstrategy. This implies that in order to pinpoint potential pitfalls for the novice programmer, which might produce superbugs, we should begin by examining the major differences between human discourse and programming discourse and the areas in which difficulties are likely to arise.

Since a full analysis of human discourse processes is outside the scope of the study reported here, the following section outlines the major features of human discourse which are relevant to the programming situation.

**Human Discourse**

Human conversational discourse has two closely interlinked strands: one is associated with language understanding, the other is to do with interpretation of behaviour of participants in the discourse and attribution of intentions. Both of these strands involve making either deliberate or default assumptions, a process which, for the large part, is unconsciously undertaken. For example, it is normally assumed that participants are able to understand natural language, can make any necessary inferences to do so, and can use appropriate doses of 'common sense' in their interpretations. Natural language interpretation involves a great deal of tacit
knowledge and inferential capacities which are taken for granted. It is also a tacit assumption that speakers in discourse have intentions and, even if the actual language they happen to use is not as precise as it might be, listeners can use their own interpretative capacities to deduce from context and prior knowledge what the intention is most likely to be.

But human discourse is also multi-faceted in the sense that not only are there rules for discourse in general, but also rules which specifically govern the current discourse. This may enable participants to use specialised language as a form of shorthand to denote specific concepts other than the ones normally designated by a particular word or phrase (e.g. 'the male gaze' in feminist discourse). Computing itself is one such 'discourse' in which the terms 'crash', 'control', 'return', 'while' and so on have specific meanings. Specialised discourses also often have associated with them sets of appropriate behaviours frequently used to consolidate the group (c.f. rugby players). Unlike more general discourse these kinds of discourse are often self-conscious - they are deliberately constructed and new arrivals may need to learn the rules and often subtle nuances of meanings which to the outsider may signify little. In first meetings with other people a great deal of effort is often put into establishing the type of discourse that this person holds, and whether it is one with which we are already familiar.

But the difficulty in assuming the sorts of co-operative conversational principles described by Grice (1975, 1978) is that, in fact, human discourse is often a relatively vague enterprise - we frequently assume that we know what we are all talking about,
but there are not so many occasions when this is rigorously put to the test. One of the ways in which correctness of interpretation can be rigorously put to the test, however, is when the recipient of information is required to act upon it. If, after having received a directive, the recipient performs what is deemed to be inappropriate action, there are two main methods of recourse. We can ask what the listener thinks was intended - assuming that the listener misheard - thereby interrupting a possibly spurious interpretation (the fault may lie either in the recipient's interpretation, or in poor formulation of the original directive). Or we may try to work out why the utterance caused such apparently unrelated behaviour, working from the assumption that the recipient is at least as well able as ourselves to sensibly interpret instructions. As participants in a complex social environment, humans are rather adept at postulating hypothetical reasons for the behaviour of others and evaluating subsequent events on the basis of those hypotheses. In this respect causal and temporal notions are very important not only for the purposes of interpreting meaning from language itself but also for understanding and relating the behavioural components of social situations. The term 'human discourse', then, involves four very closely related components:

1: the interpretation of language

2: understanding the constraints on the present discourse (if any) and what can or cannot be assumed;

3: the use of language

4: the interpretation of behaviour, and the reconciliation of actions with the content of the discourse so far.
These four components are interdependent: interpretation of behaviour is dependent on knowledge about the domain of discourse. Understanding another's actions involves (at the very least) attempting to formulate possible alternative interpretations of what has been said and generating candidate reasons to explain the observed actions. This may require the introduction of information from outside the apparent content of what has been said so far (e.g. knowing that someone is upset about some other factors than those involved in the current discourse).

Pea claims that this cycle of understanding and interpretation is applied by novices to the task of computing, and suggests that the learner's expectations are violated in programming because programming languages are formal languages underpinned by a mechanistic semantics which is often at odds with an ordinary natural language interpretation. The conflicts which arise between ordinary discourse understanding and formal discourse understanding are well documented in the study of hypothetico-deductive reasoning, and it is to this literature we turn to establish what happens when untutored subjects are asked to perform formal logical reasoning tasks.

3.2 Interpreting Formal Domains

In this section, the aim is to elaborate the sorts of strategies used by beginners attempting to make sense of a formal domain. This discussion is relevant to the study of novice Prolog programmers for two reasons. Firstly, the general argument which will be developed in the thesis is that novices find formal domains difficult to interpret, but that many errors are characteristic,
stemming from the inappropriate use of intuitive reasoning strategies. These types of errors form the basis of the generality to be observed in novice performance in different domains pointed out by Elshout et al., (1986). Such high level misconceptions are in addition to lower level difficulties experienced by beginners when trying to cope with the complexity of the computer itself, and the particularities of the language to be learnt.

Secondly, we should be especially interested in this literature given Prolog's links with logic. Prolog can be described and interpreted as an implementation of logic where emphasis is laid upon the declarative semantics, and declarative methods of problem specification. The specific argument which will be developed in this chapter is that the declarative approach to program writing can in fact mislead the novice because the formal status of expressions in the language is inadvertently disguised. Statements can look like natural language, but are underpinned by formal, frequently abstract, representations which the novice neither recognises nor understands. Thus, the student is encouraged by surface appearances to believe that intuitive strategies are adequate for the task of understanding programming.

There are two very closely related aspects to the discussion. The first is associated with the untutored subject's confusion about the interpretation of formal expressions, where natural language semantics are often used to construe their 'meaning'. The second is to do with the effects this has on the kind of problem representation and mental models that the student constructs. As far as possible it will be demonstrated that the sorts of problems described in the literature on deductive reasoning have been
observed in the computing domain, and reference will be made to relevant studies.

The deductive reasoning literature identifies several characteristic features of naive subjects' performance, which are:

1: using natural language skills to interpret logical expressions and construe 'meaning';

2: the often inappropriate use of real-world knowledge and reasoning strategies;

3: a failure to consider all the relevant information on the basis of which to reach a valid conclusion;

4: a confirmatory bias in evaluating hypotheses.

These problems will be described below, and then Johnson-Laird's (1983) theory of mental model construction is presented, which goes some way to elucidating the source of such errors. The basic difficulty for the reasoner lies in a confusion between the mental model spontaneously constructed on the basis of a linguistic analysis of problem statements, and the need to construct alternative models which allow deductive reasoning to proceed according to the rules of logic. The discussion then focuses on the difficulties for novice in the declarative approach to problem specification and logic programming. This is related to the two strands of natural discourse interpretation by novices, and the resulting conceptualisation they have of the domain in terms of the mental models they construct.

3.3 Reasoning in Formal Domains

The literature on logical hypothetico-deductive reasoning suggests that people find it difficult (Evans, 1982), and that there are obvious response biases in the range of errors made:
The most obvious aspect of these studies is the low level of logical performance. Most interesting, however, is the nature of the errors, which are far from random. (p.111)

In other words, untutored subjects' performance is not random, but it has been a challenge for psychologists to explain exactly how these particular response biases in data emerge. Attempts to do this have met with only limited success, and there is still a great deal of debate amongst researchers. Some of the reasons cited for poor logical performance are related to natural language and the interference of real world knowledge (c.f. Johnson-Laird and Wason, 1977). Certain types of logical inference (e.g. modus ponens) are reliably, and intuitively, understood by people untutored in logic, and, as Rips (1983) has pointed out, it would be very difficult to explain it to anyone who claimed not to understand it, other than by reference to the argument itself. However, there are other rules of inference which people equally reliably get wrong or find unintuitive (e.g. modus tollens). Quantified syllogisms also present many people with considerable difficulty.

The controversy amongst psychologists as to the relationship between formal, logical reasoning and 'ordinary' reasoning has a very long history. The major question is whether the human mind has an 'innate' capacity to reason in ways which are consistent with formal logic, or whether formal logic is simply one way of investigating or representing certain types of reasoning.

The rationalist view is that reasoning does proceed according to logical principles corresponding to those of classical logic, and that errors are due to misreading or misconstrual of the premises. Henle (1978) claims that there is no evidence to suggest that errors in formal reasoning tasks 'could unambiguously be attributed
to faulty reasoning'. She argues instead (Henle, 1962) that subjects in experiments with syllogisms may refuse to accept, or may misinterpret, premises therefore only appearing to reason incorrectly. Chapman and Chapman (1959) follow Henle, stating that subjects undertake illicit conversion of premises (e.g. All A are B = All B are A), so reasoning validly but with mistaken premises.

Others (e.g. Johnson-Laird, 1983) argue that classical logic is purely normative in relation to reasoning processes, and that untutored subjects ordinarily reason without recourse to it. Practical reasoning is distinguished from formal reasoning on two broad grounds: firstly, practical reasoning requires only ordinary comprehension skills, whereas formal reasoning requires analytic comprehension (discussed below); secondly, formal reasoning separates knowledge which is contained in the premises from any other kind of knowledge the reasoner may have at her disposal (Braine and Rumain, 1983). In other words, whilst practical reasoning can be performed on a mixture of information from a variety of sources (e.g. verbal discourse, general knowledge or factual discovery), formal reasoning can only refer to information contained in the premises regardless of whatever else the reasoner knows about the subject.

There are well documented examples of how difficult it can be to keep separate the two aspects of truth value (content dependent) and form (content independent). Scribner (1977) points out that in ordinary discourse they interpenetrate. One of the difficulties for beginners lies in understanding how to validate the conclusions they have reached: by reference to known facts (empirical) or by reference to a 'theory' which explicitly relates the premises to
the conclusion (hypothetical). Braine and Rumain (1983) discuss how children are frequently very puzzled as to the purpose of hypothetical experimental questions which they would resolve themselves by reference to known facts. Cole (1977) discusses similar findings in relation to Kpelle tribespeople in Liberia who put no premium on hypothetico-deductive reasoning, and who appear not to see the point of hypothetical questions, suggesting instead that the experimenter find out the answer by asking someone who 'knows' it. These are both examples of empirical validation.

But even adult reasoners who are relatively sophisticated at hypothetico-deductive reasoning by virtue of Western educational practice are not immune from being distracted from correct reasoning by the content of propositions. This susceptibility has been illustrated by Woodworth and Sells (1935) who described the 'atmosphere effect' in deductive reasoning. They argued that people are seduced by the atmosphere created by premises into accepting a conclusion which is in agreement with this atmosphere. Begg and Denny (1969) reformulated this finding to say that (a) whenever at least one premise is negative the most frequently accepted conclusion will be negative, and (b) whenever at least one premise is particular the most frequently accepted conclusion will be particular, otherwise it will be universal. The way in which the content of the task affects the ease with which correct reasoning can be performed is also frequently noted (e.g. Wason and Shapiro, 1971; Johnson-Laird, Legrenzi and Sonino Legrenzi, 1972). In these studies, it was found that the more 'realistic' the experimental task, the more accurate were the responses of subjects.

It is therefore important that the reasoner, or problem solver, be
aware that formal deductive reasoning is a very specific process which takes place only within a tightly defined frame of discourse, and, upon occasion, she may need to use what seem to be counter-intuitive methods. The frames of discourse are the formal domains normally taught in schools or university in subjects such as the sciences, or logic itself. In this view, classical logic and ordinary reasoning stand in the same relationship as the scientific discipline of physics does to 'naive physics' (Hayes, 1985).

This distinction will be important since some students participating in the studies reported in later chapters have no strong background in formal, scientific subjects - i.e. they are students who have developed good 'practical' reasoning skills, but who have had no specific training in deductive or logical reasoning. But even those students who have been introduced to the domain are quite likely to exhibit errors due to another characteristic of performance: not to consider all the necessary relevant information to make a valid judgement. This trait persists even when students are tutored in ways of making information explicit. For example, Erickson (1974) discusses the use of diagrammatic representations, and identifies three stages in deciding the validity of a syllogism:

(i) interpreting the premises
(ii) combining the premises
(iii) reinterpreting the premises

Subjects typically did not process all the available information in either one or all of these steps, which led them into fallacious reasoning. This failure was persistent even in groups of subjects who had been given training in the use of diagrammatic logic representations (e.g. Venn diagrams) which should have made
relations explicit.

It has also been noted that subjects have an almost overwhelming predisposition to seek confirmation rather than disconfirmation of their hypotheses in formal reasoning tasks (Wason, 1968; Johnson-Laird and Wason, 1970). Wason (1968) comments that commitment to a particular interpretation can produce almost pathological behaviour in subjects, who become fixated and dogmatic in their interpretation, a situation which almost inevitably leads to error.

These findings indicate that subjects tend to bring real world knowledge to bear in the formal domain, and do not have appropriately rigorous reasoning strategies to generate the necessary information for drawing valid conclusions, nor for considering alternative hypotheses. These kinds of inappropriate reasoning strategies are closely linked to confusions arising from the kinds of comprehension processes required to interpret formal expressions. These are examined in more detail below.

3.4 Interpretation Processes in Formal Domains

Braine and Rumain (1983) point out that although practical and formal reasoning have very different end products the steps in the deductive chain which lead to a conclusion may be similar. This can be a source of confusion for the beginner. Braine and Rumain go on to characterise the difference between ordinary comprehension and analytic comprehension as follows: ordinary comprehension has as its goal understanding what a speaker or writer intends by a statement; this type of comprehension relies upon co-operation of the sort described by Grice (1975, 1978), and is commonly used in conversation understanding and reading. Analytic comprehension does
not always conform to such co-operative principles: it is a process which sometimes deliberately ignores the intention of the speaker (or writer) and examines instead the extent to which the sentences permit some other interpretation. This kind of comprehension is used for specialised purposes, for example, in scrutinising legal documents, or contracts. It is less commonly used by most people and is a sophisticated skill requiring instruction and practice. People can be led astray by assuming an ordinary interpretation and making unwarranted assumptions. These assumptions often derive from natural language interpretation.

For example, sentences in natural language frequently 'invite' further inferences. Geis and Zwicky (1971) illustrate such invited inferences, where a statement such as 'if p then q' invites the inference 'if not p then not q'; 'some F are G' invites the inference 'some F are not G'. These inferences are not associated with the logical meanings of words such as 'if' and 'some': they are rather part of the process of 'co-operative understanding'. In normal discourse, the drawing of such inferences could be prevented if need be by the addition of extra clauses: 'if p then q, but if not-p then q may be either true or false'. However, of course, in the analytic mode these inferences are not available, and do not need to be explicitly cancelled.

Similarly, people often think that logical connectives have the same connotations as those in natural language. Erickson (1974) discusses 'some':

'Some' in logic does not mean what it does in natural language. In logic 'some' means 'at least one and possibly all'; it does not mean 'some, but not all'. Thus a statement like 'some dogs are animals' is perfectly all right in logic
even though a stronger statement could be made.

It is unlikely that a non-logician would consider the sentence 'some dogs are animals' meaningful. Such expressions might be discarded as 'silly' even though logically it might be necessary to consider them in deciding the validity of arguments or propositions.

Evans (1982) discusses the difference between the natural language semantics of 'if' and the use of 'IF' in logic:

Logicians often use the conditional sentence 'If p then q' to express implication, with the modification 'If and only if p then q' when it is extended to equivalence. In actual linguistic usage, however, people again tend to use the shorter form and let semantic factors determine the meaning which is read. (p.118)

Once 'semantic factors' begin to make their presence felt, then subjects may fall into the trap of believing that what is sensible is correct, and what is true is valid. Understanding that an inference schema (such as modus ponens) is always valid, but leads to true conclusions only if the propositions substituted into it are true, can present problems for beginners, particularly in the light of the previously noted prediliction people have to refer to real world knowledge.

In truth conditional languages, unlike natural languages, sentences are only either true or false. However, many people would regard some propositions as 'neutral' or 'void' with regard to truth value. For example, if John has no children, then statements such as 'All of John's children are asleep' are neither true or false, but void or irrelevant (Johnson-Laird and Wason, 1977).

A further difference is that truth conditional languages do not
directly deal with temporal or causal events. For example, many people would attribute different meanings to the following sentences:

Hostages were taken and the President bombed Libya

The President bombed Libya and hostages were taken

though they are identical in the propositional calculus, since it does not accommodate causal or temporal relations. However, there is both a temporal and a causal reading to the above sentences which, from an intuitive standpoint, seem to be the crux of their 'meaning'.

A possible solution to this kind of difficulty is to keep the material abstract, and only use symbolic logic - i.e. remove natural language as far possible. This can be equally problematic, however, for two reasons. The first concerns the nature of abstract representations: we have already mentioned some studies which indicate that performance is improved if context is given to the task (Wason and Shapiro, 1971; Johnson-Laird, Legrenzi and Sonino Legrenzi, 1972). Keeping material abstract can, therefore, make reasoning tasks unnecessarily difficult.

The second reason is that even if the syntactic interpretation of logic is adopted, beginners will find it very hard not to try to convert things into natural language in order to understand them. The idea that symbols can be shunted around with no apparent concern for what they mean, and still produce a correct answer is a relatively sophisticated one, and presupposes a certain amount of confidence which the learner may not have. It also reflects the
difference between degrees of expertise - experts may find it easy to say which parts of a problem/expression/technique can safely be ignored, but beginners may not.

In this brief overview some important factors which affect successful performance in the domain of deductive reasoning have been discussed. Untutored subjects tend to use real world knowledge to inform their reasoning in formal domains, and are inclined to use ordinary interpretative processes in understanding logical expressions, rather than analytic processes. This means that statements are not properly scrutinised to discover what other possible interpretations they may have, a process which allows the reasoner to generate sufficient information to make a valid deduction. Verification rather than falsification of hypotheses would seem to be linked to this tendency, because the process of falsification depends upon analytic interpretation of one's own hypotheses. Johnson-Laird's (1983) discussion of mental models helps in understanding how all these factors affect problem conceptualisation, and problem solving.

3.5 Mental Models and Natural Language Interpretation

After long acquaintance with the field of deductive reasoning Johnson-Laird (1983), by adopting a model-based view of natural language discourse comprehension, has developed a theory of how logical syllogisms can be reasoned about within the framework of mental models, rather than from some kind of 'mental logic' using formal rules of inference. The main contribution of his work for our purposes is to elucidate the differences between natural language and formal language interpretation, according to the
principle of model construction, and to illustrate how the process of formal reasoning can be adversely affected by linguistic factors.

Before entering into the discussion, an important point should be made. There is a very close relationship between the kind of mental models of situations or discourse created spontaneously by learners, and the kind of model one might present in a teaching situation to assist in the process of mental model construction. This point is emphasised by de Kleer and Brown (1982):

.... properties that make... mental models learnable or that facilitate their development turn out to be essential characteristics of a highly robust mental model in the first place. (p.285)

It will be made clear in the following discussion which kind of model is being referred to, but at times the distinction can be blurred.

Johnson-Laird reiterates the difference between the automatic, unconscious 'implicit' inferences required to understand natural language discourse, and the 'explicit' inferences needed for logical reasoning. He claims that the main distinction between them is whether or not there is a deliberate search to find alternative models for the discourse.

Ordinary discourse, which is often ambiguous or incomplete, is understood (using implicit inference processes) by creating only one mental model of the ongoing state of affairs being discussed, though this model will be refined as further information is received. However, in order to accomplish this, some necessary assumptions are made:
A description of a single state of affairs is represented by a single mental model even if the description is incomplete or indeterminate... In theory, the single model should be constructed by a non-deterministic device that always produces the correct model; in practice, non-determinism has to be simulated by a procedure that constructs an initial model on the basis of plausible, though sometimes arbitrary, assumptions and recursively revises the model should such an assumption turn out to be wrong. Hence, the content captured in a mental model - its significance - is a function of both the model and the processes that evaluate it. (p.408) (my emphasis)

Examples of this process are given in terms of how an initial model turns out to be incorrect, as in the sentence 'The horse raced past the stable fell'. Making plausible assumptions, therefore, is a necessary part of natural language comprehension and mental model construction.

However, in the process of making explicit logical inferences, learners need not only these interpretative skills to construct a mental model of the state-of-affairs described, but they must also understand the 'fundamental semantic principle underlying valid deduction':

... an inference is valid if and only if there is no way of interpreting the premises that is consistent with a denial of the conclusion. (p.98)

Thus, the learner must search for alternative models of the premises to check for validity. Implicit inferences associated with natural language interpretation are therefore very likely to be invalid if used in the formal domain, since no alternative candidate models of interpretation are sought. The above principle of deductive reasoning, however, is frequently not recognised by many learners and, left to their own devices, they are unlikely to intuit it. Johnson-Laird points out that the mental models used to
construct 'meaning' out of ambiguous natural language discourse
incorporate general knowledge and knowledge of context by default:

.... knowledge is embodied in the model by default, that is, it is maintained in the model provided there is no subsequent evidence to overrule it. No attempt is made to search for an alternative model unless such evidence arises. (p.128)

The important ramification of this is: if learners are unaware of the nature of formal reasoning, and use implicit inference processes to construe the formal domain, in what way could they be challenged in their single interpretation? Their model will only be revised if assumptions turn out to be wrong. But learners can only perceive that they are wrong if they already understand the constraints of the formal domain and recognise the 'underlying principle of deduction' - i.e. that they should construct more models. But how many more? Given the tendency to verify and confirm hypotheses, even if subjects have been acquainted with this principle, there is no guarantee that they will necessarily construct sufficient models to check for validity.

This is an extremely difficult situation to remedy, and the question of how to help beginners in this respect is still largely unresolved. It has been noted that Venn diagrams, which theoretically should help identify other possible interpretations, do not appear to enhance performance (Erickson, op. cit.).

Johnson-Laird's interpretation of this finding is that diagrammatic representations of logic (such as Venn diagrams) are unsuccessful in helping students reason because they violate the principle that models should have similar 'relation structure' to situations or processes that they model. That is, the relations and structures
of the external situation to be modelled must be reflected in the mental model. If this relation structure does not exist then the model simply mimics the phenomenon, and cannot be used explanatorily, or for effective reasoning.

Johnson-Laird identifies the constraint of economy to give the following principle of structural identity:

The structures of mental models are identical to the structures of the states of affairs, whether perceived or conceived, that the models represent. (p.419)

This fundamental property of mental models is also stressed by other researchers (e.g. Norman, 1983; de Kleer and Brown, 1981). The principle means that certain types of representation will often not be successful as tools to assist in mental model manipulation - e.g. propositional representations, semantic networks or logical expressions - on the grounds that there may be no isomorphism between the representation and the situation.

The relevance of Johnson-Laird's discussion of mental model construction to novice programmers is that it lends coherency to the behaviour of beginners who are trying to construe some sense in a formal domain. The effect of linguistic interpretation on task conceptualisation, as has been pointed out, is liable to be extremely powerful, particularly given the very different approaches required for ordinary reasoning and formal reasoning. Ordinary reasoning seeks to confirm hypotheses generated from an incrementally constructed mental model. Formal reasoning, on the other hand, depends on the ability to construct alternative models from the same stimuli. Naive reasoners are concerned simply to 'make sense' of what they have been given, leaving many important
aspects of the problem domain unexplored. Natural discourse interpretation involves many implicit assumptions which are taken for granted, but which, in the formal domain, would have to be made explicit.

An empirical study in the programming domain which illustrates the effect of mistaking formal discourse for natural discourse is that by Miller (1981). He was investigating the feasibility of using natural language as a programming language, and reports on an apparent inability of non-programmers to specify a task in English sufficiently precisely for 'some other person' to perform the task from the instructions.

The problem involved three types of data files kept on employees in a business, each file containing different types of information. Subjects were given six procedures for finding information about employees, some of which required cross-referencing files. Instead of answering the questions themselves, they were asked to write a list of instructions to enable someone else to obtain the necessary information.

Subjects first expressed reluctance to perform the task indicating that they were used to following rather than specifying instructions, and, besides, they claimed, 'the problems were straightforward and required little explanation'. When the results were examined for completeness and correctness, though, Miller comments:

It was the case that most solutions were almost never really clearly incorrect; either they were obviously right - albeit incomplete - or else they were sufficiently incomplete that one could generously imagine omitted steps which, if supplied, would render them correct. (p.206)
The first notable point in Miller's analysis is that 'natural' inferencing processes were regarded as sufficient to make the task explicit, and did not require any further elaboration. The need to plan a solution appeared to be ignored: subjects leapt straight in, and began typing within a very short time of reading the problem, showing no evidence of having planned the solution beforehand. What planning did emerge was of a localised nature, subjects proceeding in a linear and incremental way. Not unnaturally, as problem complexity increased, completeness decreased.

Miller is uncertain whether difficulties were expressional or conceptual - i.e. people could solve the problem but were not used to using English in such a formally expressive way, or that they could not conceptualise the problem properly which was why it was badly expressed. He suggests that his subjects did not properly understand or deal with the 'subtle conceptual complexities' of the problem, and concludes on the sceptical note that natural language as a programming language would allow people easily to create programs that did not do what they were supposed to do. Whilst this conclusion seems appropriate, given the specialised nature of the task, it seems unfair to suppose that subjects were unable to perform well, or that they failed to appreciate 'subtle' complexities. The task provided was ambiguous in the sense that natural language was the only available language for these subjects, and Miller scrutinised it as if it were an unspecified formal language.

The findings in this study in a sense capture the very difficulties that novice programmers encounter. Given a problem, it is not sufficient to solve it oneself in whatever way seems possible. The
demand is to find a form of description which circumscribes the task in a formal sense — i.e. one which makes explicit all that may be implicit in the natural language description. The resulting instructions must then be scrutinised analytically to discover whether there is room for alternative interpretation. In this respect, both formal reasoning, and task specification, require an extra level of interpretation above the ordinary — to consider various ways in which statements could be interpreted, other than the way they were intended. This level of abstraction is difficult, particularly if one is not already familiar with the task in question.

Miller's subjects thought that ordinary natural language interpretation would be sufficient because there was no reason why they should think otherwise. People are not unnaturally inclined to assume that statements in natural language are explicit enough unless they are made aware of how much more explicit they need to be for use in formal domains, where the semantics are impoverished in comparison with those of natural language. This experiment will used in the following chapter to illustrate the discourse framework.

There have been two major strands in the discussion so far: the first is associated with the relationship between ordinary and analytic interpretation of expressions; the second is related to the process of mental model construction which results. We now move on to consider claims that have been made on behalf of Prolog (Kowalski, 1979) and to show how these are misleading by reference to both these lines of argument.
3.6 Logic, Declarative Semantics and Natural Language

Some of the arguments which have been used to support Prolog as a programming language stress the relationship between logic and Prolog, and use this as its main claim to 'learnability' and ease of use. This argument pivots on the relationship between logic and natural language, and needs clarification, since it could easily be misinterpreted by beginning programmers, or by teachers, who may not have strong backgrounds in either logic or computing against which to evaluate the claims.

Kowalski (1973; 1979) argues that logic programming languages are higher-level than other currently used programming languages because predicate logic as a programming language has a syntax and semantics which can be understood in terms of classical logic. In other words, he claims that because such languages are based on the 'rationalisation of human thought processes' (i.e. predicate logic) and not on the kind of logic which underpins 'machine operations', they have a very particular kind of machine-independence:

The meaning of programs expressed in conventional languages is defined in terms of the behaviour they invoke within the computer. The meaning of programs expressed in logic...can be defined in machine-independent, human-oriented terms. As a consequence, logic programs are easier to construct, easier to understand, easier to improve, and easier to adapt to other purposes. (Preface)

In theory, then, this means that programs can be fully described in terms which are nothing to do with the actual machinery on which they run, and they need not reflect any facet of the underlying architectures of computers. Furthermore, because the language of logic can be used as a specification language and as a programming language (at least in theory), the process of program development
is streamlined by eliminating successive translations from one representation to another, and allows for more effective deployment of programmer time and effort.

In practical terms there may be great advantages for the expert programmer in using logic programming languages. However, the line of argument which is contentious is the one which says: since predicate calculus is a formalisation of the logic of natural language and rational human thinking processes, logic programming is not only more 'human-oriented' than other programming formalisms, it is more 'natural' and, furthermore, can be understood in terms of its natural language equivalents (Kowalski, 1979).

The above quotation, and the following claim are to be found in Kowalski (1979), a book which explicitly states that no previous knowledge of logic, problem-solving, or computer programming is assumed:

.....when programs are expressed in symbolic logic, they can be understood in terms of their human-oriented, natural language equivalents. (p.9)

The point of confusion lies in an assumption that people will necessarily find logic easier than any other programming language on the basis of its relationship to natural language. All the arguments so far presented suggest the complete reverse. Unless learners know about the underlying models of the formal domain, they will use ordinary discourse comprehension strategies to understand programs. It is quite right that beginners should be able to understand programs without having first to come to terms with detailed concepts associated with complex machinery. But this
link with natural language is misleading. Besides, the kind of language which is used in describing logical expressions is not the same kind of natural language that one might use in ordinary discourse - i.e. the underlying model is that of logic, not natural language semantics, and it therefore requires an analytic reading.

Kowalski (1979) does make the caveat that natural language will only provide an 'informal guide' to understanding logic. This is appropriate for someone who understands either logic, or programming, or both. He or she will have a fundamental grasp of the limitations of the 'informal guide' as an aid to understanding, and will be able map logical expressions onto the correct domain model.

However, the interpretation which the inexperienced novice is liable to make is that logic programming is more natural than any other programming formalism and is therefore easy, and that 'natural language' skills will be sufficient to interpret and write programs. We have already discussed the kinds of conceptual problems beginners have in distinguishing formal from practical reasoning in the domains of logical reasoning and practical problem solving. Errors in deductive reasoning are frequently linked to a failure to distinguish between formal languages and natural languages. The logical component of Prolog is as likely to be misinterpreted as standard logic. In fact an even stronger claim could be made specifically in relation to the declarative (non-procedural) interpretation of Prolog which is as follows: in the absence of procedures which give clues as to how something is to be interpreted, beginners are likely to make heavy use of natural language interpretation skills because they have no choice to do
otherwise. This may have adverse effects on the ability of students to write effective programs, and to decompose problems satisfactorily. Some studies are described below which illuminate the divergence between intuitive natural language interpretation and strategies appropriate to declarative problem specification. First, 'declarative' programming is defined.

3.7 Declarative Programming and Problem Specification

Baron, Szymanksi, Lock and Prywes (1985) describe and defend declarative (non-procedural) programming as follows:

Programming with a non-procedural language requires focusing on analysis of the problem rather than on a solution in the form of a sequential computer program.... the non-procedural language fits better into the natural process of problem solving. The user of a non-procedural language specifies a set of constraints to be satisfied which are derived from the problem, rather than a sequence of steps to be taken. (p.127)

They argue that declarative constraint specification is a part of 'natural problem solving' because it defines the goal to be achieved. To support the need to develop non-procedural languages, they further argue that people are often unnecessarily 'forced' into a procedural mode of problem solving with computers because such procedurality reflects the inner workings of the machine, and is therefore convenient. They suggest that the role of computers in the future should be to 'translate non-procedural specifications into procedures' thereby eliminating a great deal of work for the programmer. Since achieving this state-of-affairs is one of the goals of the logic programming community, this point will be discussed in relation to Prolog.

There are two problems with this view, both associated with the
interference of ordinary natural language comprehension processes. The first relates to the observations discussed earlier where people typically do not consider all the relevant information in formal problem solving. Miller (1981), described above, illustrates how implicit assumptions in ordinary discourse prevented untutored subjects from properly specifying tasks in English; that is they assumed that the discourse they were entering was ordinary human discourse, and that ordinary interpretations of natural language would be sufficient. The second is related to studies by Ehrlich, Soloway and Abbot (1982) which demonstrate a tendency for people to regard formal expressions as 'merely' descriptive, leading to incorrectly formulated expressions.

Soloway, Lochhead and Clement (1982) and Ehrlich, Soloway and Abbott (1982), in related studies, investigated the possible enhancement effect of computer programming on algebra word problem skills. It had been noted that ostensibly simple algebra word problems caused American college students a great deal of difficulty and a correspondingly poor performance rate. An example problem is:

Write an equation using the variables $S$ and $P$ to represent the following statement: 'There are six times as many students as professors at this University'. Use $S$ for the number of students and $P$ for the number of professors.

Results:

Sample size: 150 Correct answers: 63% Incorrect answers: 37%

Other problems of the same or similar form produced much higher error percentages. Students in the 'incorrect' category typically represented the above equation as follows:
6S = 1P

Soloway et al. argue that the source of the error is that subjects interpret the reversed equation above as representing a 'passive' state of affairs: that there are 6 students to 1 professor, and the equals sign expresses a comparison rather than an equivalence. The correct equation \((S = 6P)\) represents instead an 'active' operation being performed on one number (the number of professors) to obtain the second number (the number of students).

In other words, the equation \(S = 6P\) is not a direct description of the actual situation, but rather, it represents the hypothetical state of affairs which would result after performing the operation of multiplying the current number of professors by 6. . . . the equation is thus interpreted in a procedural manner as an instruction to act. (p.175)

Soloway et al., and Ehrlich et al. found that when the problem was presented in a programming context, students had less difficulty getting the correct response because they were in a 'procedural' environment which cued them to think of inputs and computations rather than static descriptions of states-of-affairs.

But Baron et al., arguing in defence of non-procedural languages and rebutting Ehrlich's conclusions, claim that algebra:

. . . . can...and usually is used to represent constraints on solutions, independently of whether the user knows in advance what procedures will be used to solve the equations thus formed. (p.129)

It is true that the user may not know yet precisely which procedures will be used, but she must nevertheless be aware that there are procedures which eventually will be applied. Ehrlich's point is that such procedures are not foremost in the minds of students away from the programming domain. We would argue that
these implicit procedures are not part of the natural language interpretation domain, but are components of the model underlying the formal domain. Since they are not cued in the representation, it is no surprise that some students misperceive the problem, and use natural language statements in a descriptive manner.

We have discussed issues related to the kinds of misapprehension that beginners can have of formal expressions, which are a result of using ordinary comprehension to interpret formal expressions. In the next section the focus is on how novices conceptualise problems from problem statements, and, in particular, the way in which the problem solver's mental model of a task is influenced by applying ordinary discourse interpretation processes to problem statements, when those problems are to be solved within a formal framework.

3.8 Mental Models and Problem Interpretation

The point was made earlier that effective mental models are those which have direct correspondences to the situation being modelled. It follows that if a problem statement itself promotes a clear concrete representation then problem conceptualisation, and problem solving, will be easier.

The literature on human problem solving has established that more effective performance is achieved when concrete representations for problems are used prior to the use of more abstract formulations which may describe the solution to a class of problems (e.g. Mayer, 1981). The use of a concrete model clearly limits the space of possible alternative interpretations, particularly if objects, relations and 'moves' are made explicit. More importantly a concrete model can, to a large extent, compensate for the lack of
analytic comprehension in beginners.

The extent to which the problem solving process can be disrupted by linguistic factors which disguise underlying concrete models is illustrated in a study by Hayes and Simon (1974). They conducted an experiment in problem comprehension, postulating two major processes in the initial stages of reading a problem statement:

1) Language process - where the problem solver reads short segments of text and extracts information from them through syntactic and semantic analysis

2) Construction process - in which the newly extracted information is checked against and added to the subject's developing model of the situation

By iteration of these two processes the subject firstly understands through linguistic analysis what the problem says, and secondly interprets this information to incrementally create a problem space. Problem spaces represent an individual's conceptualisation of the task, and correspond to mental models. The structure of the problem space will determine not only the kind of solution obtainable, but also the effectiveness with which it is found.

Hayes and Simon hypothesise that having once constructed a problem space for a problem of a given type, the problem-solver should be able to use elements of that space in isomorphic problems, giving him or her the advantage over a problem-solver who must construct the space from scratch. However, they were interested particularly in the ways in which recognition of a previously constructed problem space could be hampered by linguistic factors in the presentation of the problem statement.

Their experimental subject was given The Himalayan Tea Ceremony a
problem isomorphic to the Towers of Hanoi problem (Figure 3.1), i.e. the only differences are linguistic.

The Himalayan Tea Ceremony

In the inns of certain Himalayan villages is practised a most civilized and refined tea ceremony. The ceremony involves a host and exactly two guests, neither more nor less. When his guests have arrived and have seated themselves at his table, the host performs five services for them. These services are listed below in the order of the nobility which the Himalayans attribute to them:

- Stoking the Fire
- Fanning the Flames
- Passing the Rice Cakes
- Pouring the Tea
- Reciting Poetry

During the ceremony, any of those present may ask another, "Honored Sir, may I perform this onerous task for you?". However, a person may request of another only the least noble of the tasks which the other is performing. Further, if a person is performing any tasks, then he may not request a task which is nobler than the least noble task he is already performing. Custom requires that by the time the tea ceremony is over, all of the tasks will have been transferred from the host to the most senior of the guests. How may this be accomplished?

The Towers of Hanoi

Somewhere near Hanoi there is a monastery whose monks devote their lives to a very important task. In their courtyard are three tall posts. On these posts is a set of sixty-four disks, each with a hole in the centre, and each of a different radius. When the monastery was established, all of the disks were on one of the posts, each disk resting on the one just bigger than it. The monks' task is to move all of the disks to one of the other pegs. Only a single disk may be moved at a time, and all the other disks must be on one of the pegs. In addition, at no time during the process may a disk be placed on top of a disk that is smaller than it. The third peg can, of course, be used as a temporary resting place for the disks. What is the quickest way for the monks to accomplish their mission?

N. B. Students are normally only expected to solve the problem initially using three or five disks.

Fig. 3.1 The Himalayan Tea Ceremony and the Towers of Hanoi

For the Towers of Hanoi, the problem solver must end up with three pegs, and five different sized disks. For the Tea Ceremony, the
problem solver must end up with the following objects and relations: three participants, five tasks, and an ordering on the nobility of the task. The order in which the tasks must be 'moved' is indicated by the relative seniority of the guests - from the host to the senior guest via the junior guest. This problem statement does not promote an obvious model of the sort that underlies the Towers of Hanoi. In the Towers of Hanoi, the objects in the problem correspond to real objects - disks and pegs - and the size of disk is a physical attribute. In the Tea Ceremony the objects are people and tasks, where the ordering on tasks is associated with the concept of nobility. Even if the subject uses a diagram to assist in model construction, how best should a 'noble task' be represented?

The effect of this presentation was such that the subject did not recognise the isomorphism, and therefore had to construct the problem space afresh, despite his supposed advantage. Furthermore, the authors note, he did not find it at all easy. This finding implies that people typically do not reduce problem statements to some kind of basic structural form (i.e. abstract problem spaces) but generate a new model on the basis of the objects and relations referred to in the problem statement. This also emphasises the fact that if some knowledge is relevant to a problem solving episode, students cannot be relied upon to spontaneously bring it to bear, unless they have been cued to do so. Otherwise, they will rely only on their ordinary interpretation of the problem statement.

There is another aspect to this study which has not yet been mentioned: the Tea Ceremony is a declarative form of the more overtly procedural Towers of Hanoi. The effectiveness with which
beginners can learn and use declarative representations is clearly relevant to an analysis of Prolog novices.

3.9 'Correspondence' Models and Declarative Representations

Difficulties in understanding the declarative method lie in the difference between the kind of initial mental representation of the problem domain created by the learner from the problem statement, and the kind of representation which is appropriate for use in declarative programs. It will be argued that due to ordinary comprehension processes being applied to problem statements, beginners will find it easier to conceptualise problem situations if the problem statement supports a clear structural model which can then be used by the novice as a basis for reasoning. The emphasis for the novice will be upon the moves which can be performed to achieve a solution, and in this sense, primary problem solving procedures are likely to be procedural in character.

Given the 3-hoop Towers of Hanoi problem, the initial representation which a novice would be likely to generate, using ordinary comprehension processes, would be a variant on Figure 3.2. This is a structural model, having direct correspondence to the objects mentioned in the problem statement, which can be used to work out the sequence of moves for hoops to achieve a solution. We will call these kinds of models 'correspondence models' because they do not require extra levels of analysis beyond that of interpreting the problem statement.
Let us take another problem, the River Problem (Figure 3.3).

A farmer is by a river with a wolf, a goat and a cabbage, and wishes to cross to the other side. There is a boat which will hold only two 'objects' at a time. Only the farmer knows how to row the boat. He must move all three items from the left to the right bank without leaving the wolf alone with the goat or the goat with the cabbage. There are obvious gastronomic reasons for this stipulation. Note that the wolf is not particularly partial to cabbage, and may be left alone with it.

Again, through ordinary interpretation, the novice is liable to end up with a correspondence model similar to that in Figure 3.4. - a structural model

which corresponds to the state of affairs, and which allows the problem-solver to work out the necessary moves. Having initially solved the problem using this kind of model, the learner is then ready to begin refining the solution, and finding ways of expressing it in the terms of a programming language. Emphasis is likely to be on expressing the move sequences, the basic model remaining the same. The task is then to find an appropriate algorithm to express it. That beginners tend to work in this
algorithmic style has been noted by Van Someren (1985), who observes it in Prolog novices with no previous experience of programming.

But now consider a declarative program for this problem (Figure 3.5):

State(x, y, u, v) expresses that there is a state in which the farmer, wolf, goat and cabbage are on banks x, y, u, v respectively of the Thames. Initially they are all on the north bank with a boat and want to cross to the south:

State(N, N, N, N). and State(S, S, S, S)?

Opp(u, v) expresses that u and v are opposite banks and Safe(x, y, u, v) expresses that the state in which the farmer, wolf, goat and cabbage are on banks x, y, u, v respectively is safe. The farmer can cross from one bank to the other taking at most one passenger provided the new state is safe:

State(x, y, u, v) if State(x', y, u, v)
and Opp(x, x') and Safe(x, y, u, v).

State(x, x, u, v) if State(x', x', u, v)
and Opp(x, x') and Safe(x, x, u, v).

State(x, y, x, v) if State(x', y, x', v)
and Opp(x, x') and Safe(x, y, x, v).

State(x, y, u, x) if State(x', y, u, x')
and Opp(x, x') and Safe(x, y, u, x).

Opp(N, S).
Opp(S, N).

A state is safe if the farmer and goat are together on the same bank, so the farmer can protect the goat from the wolf and the cabbage from the goat. It is also safe if the farmer and goat are on opposite banks, but the wolf and cabbage are together with the farmer:

Safe(x, y, x, v).
Safe(x, x, x', x) if Opp(x, x').

Fig. 3.5 Declarative program for the River Problem
(Kowalski 1982, p.10)

We note that although this is a complete specification of the
problem, it is not based on a structural model of the problem situation. Instead a particular representation has been exploited, one which capitalises on the use of variables. Even for a non-novice, it is difficult to see what is 'going on' - i.e. who has moved from where to where taking what with them? Where have all the objects gone? This is a rather extreme example - there would be other ways to write this program which were more overtly informative with regard to moves, and beginners would not be expected to either understand or create such a representation initially. But the basic point is that declarative problem solving is concerned almost exclusively with exploiting representations, aiming for generality and completeness. In this respect, most of the important work lies in the process of deciding upon a good representation. Having accomplished this, the actual writing of the program is indeed almost trivial - it just 'falls out' of the representation. Now although it is equally true that more conventional procedural languages also aim for generality, the kinds of representations which underpin them can be mapped back, at some level, to a correspondence model. Learners therefore have the opportunity to incrementally develop their representation skills until they are ready and able to use more abstract, declarative-style representations.

This issue of representation is very important, and will have to be confronted by the learner at some stage. But, initially, declarative representations are going to be difficult for novices to 'unpack' because they do not easily decompose into structural models which the beginner can recognise. Objects and moves are frequently hidden deep inside the representation which may bear
very little resemblance to the original problem statement. Given that novices are very much influenced in the initial models they construct by surface features of problem statements, they cannot necessarily be expected to recognise the relationship between such a representation and the problem. The major point is that the underlying mental model of a problem must change in order to effectively use declarative methods, and this in itself represents a new problem for beginners, an extra layer of interpretation.

The discussion so far has concentrated on interpreting problems in formal domains. It has been emphasised that the novice's conceptualisation of the domain, expressions in the domain, and of the particular problem to be solved are all heavily influenced by ordinary comprehension processes which are normally used in natural discourse. A major component of this situation is the lack of analytic comprehension in untutored subjects, who tend not to recognise that the implicit assumptions and inference processes which allow them to understand natural language are not part of the formal framework, and therefore cannot be assumed. But there is one facet of novice behaviour in deductive reasoning which has not yet been addressed: the tendency to confirm hypotheses rather than disconfirm them. This modus operandi has been also observed in the computing domain, and has been dubbed 'abductive reasoning'. This issue is discussed in the following section, where it is emphasised that the other important component of ordinary discourse is the interpretation of behaviour as well as language. This is an important consideration in the study of novice programmers, since part of their task is to predict, interpret and explain the behaviour of the computer.
3.10 Abductive Reasoning

Mack, Lewis and Carroll (1982) studied novices trying to learn how to use text-editors from instruction manuals and self-study materials. Two related sets of difficulties were identified: those associated with the editor itself, and those related to the user's strategies.

In the first category it was noted that excessive jargon in texts confused subjects; that interface features were not always obvious and consistent; and that 'help' facilities often made assumptions about the abilities of learners to describe their own difficulties.

In the second category they discovered that their subjects made ad hoc interpretations of situations, constructing explanations for what they saw. These explanations sometimes prevented subjects from realising that they did in fact have a problem. Learners tended to generalise from what they already knew (or thought they knew), and they did not always read or follow directions. Even when they did attempt to follow instructions they frequently misinterpreted them. Furthermore, their subjects seemed not to realise the inter-dependency of some of their difficulties - i.e. having resolved one aspect of the problem, they were surprised to find another had taken its place.

Lewis and Mack (1982) went on to 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 help them to understand what was happening were able to generate interpretations, and these
interpretations formed a major part of what they learned, rather than anything contained in texts. This process, they suggest, resembled 'abductive reasoning' (Peirce, 1958) which is described as follows: if some observation O is implied by an assumption A, together with beliefs already held, then A is adopted. Abduction is seen as 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. (p.2)

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.

There are certain problems which abductive learning brings with it. Firstly, learners frequently do not realise anything is wrong either with their interpretation, or with the system itself. Secondly, superficial resemblances between things the learner thinks she needs and what she sees available often lead to the use of irrelevant information. In other words, in the search for some tool or strategy, the learner will pick up anything that appears
to satisfy the need, whether or not it actually does. Thirdly, learners often do not test their abductive hypotheses by, for example, deductive exploration. This leads to only partial interpretation of information, since abductions are generated only to explain odd events rather than being part of an integrated understanding process.

Lewis and Mack conclude that abductive learning is a 'cheap' but powerful way of creating new knowledge because, when the abduction is correct, a generalisation is made with very little effort. The authors emphasise that, often, abduction is the only available method of proceeding in complex learning situations because given information is incomplete and ambiguous. Since abductive reasoning processes often relate new situations to known information, it is tempting to assume that it is a form of learning-by-analogy. Lewis and Mack point out, however, that whilst learning-by-analogy takes place when the old and the new knowledge are both clearly identified and links are systematically made between the two, abduction takes place from one point only (the present observation), and works out toward what the learner (for whatever reason) thinks is relevant:

The learner is free to link up whatever is encountered [in memory] rather than having to develop an explanation of certain given facts. (p.7)

These observations are consistent with findings in the deductive reasoning literature, and buttress the claim that learners may not even attempt to disconfirm or falsify their theories. Having abducted an assumption, of course, future abductive reasoning will be based upon that 'theory', rather than upon a correct model. In
this way, abductive reasoning can also play a role in preventing the learner from considering all the necessary relevant information, and can contribute to a tangled web of misperceptions.

But it is crucial to realise that beginners who do not yet have any appropriate domain knowledge have no choice but to interpret and construe new information, tying it in with existing knowledge, and forcing a fit where necessary. These are often perfectly good learning strategies. However, they can occasionally lead beginners astray. Whilst many people can - and do - recover from early misperceptions, some learners become trapped, and find themselves unable even to describe their own problems, let alone navigate their way out of difficulties. It should also be obvious by now that certain features of a language may dispose beginners towards misperceptions (c.f. the discussion about declarative interpretations in the previous section). To help beginners, therefore, it is essential to analyse these very high-level misconceptions in order that future developments in computing languages avoid certain kinds of features which at first sight might appear to be advantageous to the beginner but which turn out in the end to be red herrings.

3.11 Conclusions

The naive strategies of novices in formal domains have been elaborated. The overall picture emerges of learners attempting to construe a formal domain using natural language interpretation skills which lead them to only apply ordinary comprehension strategies. From this they construct a single model of the
discourse, which will be revised only on perceiving contradiction. This model will contain various default assumptions, and will be used to generate plausible hypotheses which the learner will attempt to confirm. Confirmed hypotheses become part of the learner's conceptualisation of the task, whether they are correct or not. The ability to successfully construct a mental model is highly influenced by linguistic features of the problem statement, which can frequently prevent the learner from bringing relevant knowledge to bear on the current situation.

Natural discourse has an obvious salience over formal discourse. It is the one with which we are all familiar, and formal domains do not have the same richness of interpretation and content. Almost anything can be talked about and represented in the natural domain, but not so in formal domains where the boundaries between what is or is not legal are often sharply drawn. Formal 'languages' are, in any case, representational schemas which are necessarily impoverished in comparison with natural language, not only in terms of structure, but also in terms of usage - we do not talk in logic, or Prolog. Difficulties become acute, though, when students have no underlying domain models for interpreting expressions in the formal domain, because, in this case, tacit assumptions will not be challenged or brought to the attention of the learner. People using ordinary comprehension processes can mistakenly believe that they have 'understood' a problem, not recognising the insufficiency of their mental model (c.f. Miller 1981).

In the discussion of Johnson-Laird's (1983) theory of reasoning, the complexities of the interaction between natural language
interpretation processes producing models of discourse, and mental model construction for formal logical reasoning were noted. Language interpretation normally requires a single model of the discourse which is revised only upon perceiving error, whereas formal reasoning requires that multiple models be created in order to make valid judgements.

In effect, Pea (1986) restates some of these difficulties by suggesting that a major source of difficulties for novice programmers lies in the fact that they tend to treat programming discourse as analogous to human discourse. The main problem with this tendency is that ordinary human discourse is not governed by the same sorts of constraints as those governing the formal domains of logic or programming, and the language of the discourse is natural language. Pea's view subsumes the kinds of problems identified so far in relation to language understanding, but also emphasises that novice programmers are dealing with the behaviour of the computer which complicates matters still further.

In this chapter we have described and illustrated the sorts of strategies which novice programmers are likely to bring with them, and which constitute their contribution to the learning situation. These strategies cannot be ignored either in teaching materials, or in the analysis of novices' difficulties. The next chapter describes a framework which draws together the findings discussed in this chapter, and which provides a map of the space of potential errors into which beginners may fall.
CHAPTER 4

A Three-Levelled Discourse Framework for Prolog

4.1 Introduction

In Chapter 2 it was argued that novice programmers are not 'empty vessels' but have general purpose problem solving and learning strategies which they use to interpret novel domains. In Chapter 3 it was shown that many of these strategies are based upon pragmatic considerations which are often context-dependent interpretations of real world events. So, when untutored people interpret hypothetico-deductive reasoning tasks, it seems that a major source of their misunderstandings lies in the tacit assumption that the domain of discourse is that of the ordinary world, and the language of the discourse is unconstrained natural language. People quite naturally tend to assume that the domain of discourse is that of human-human communication - i.e. that their responses will be read and interpreted by another person - and this assumption leads them to construct expressions or sentences which require ordinary (non-formal) co-operative understanding. This seems a perfectly sensible assumption to make if one is untutored in formal reasoning.

Given that this tendency exists in the performance of deductive reasoning tasks, which are usually conducted using only pen and paper, then, clearly, in the programming domain where a computer actually participates in the discourse as an 'other' contributing its own behaviour to the situation, the possibilities for this particular type of misapprehension are much greater. Because of the contributing behaviour of the machine, the problem is twofold. Learners may make assumptions about how to interpret things they
have written themselves (i.e. as they do in deductive reasoning tasks), and they may then make further assumptions about the capacity of the computer to understand and interpret what they have written. The distinction is between 'what do I think this means?' and 'what do I think the computer thinks this means?'. The first question relates to what the novice thinks she has communicated to the machine, on the assumption that the computer understands language like her, and the second relates to her predictions of the machine's behaviour, on the assumption that the computer behaves like her. As program writing progresses, the novice must evaluate the extent to which the computer's activities (a) are what she wanted, and (b) have contributed towards a solution, or the desired goal.

If it is true that novice programmers use ordinary human discourse processes to interpret the formal mechanistic domain of programming, as Pea claims, and as seems the case in deductive reasoning, at what points do things typically go wrong? The dynamic interactive aspect of human/computer discourse sets programming apart from other formal reasoning tasks, suggesting that there are liable to be difficulties which are particular to programming which are not observed, for example, in the field of deductive reasoning. Nevertheless, it still remains to be seen how many of the problems experienced by beginning programmers are linked to the fact that programming languages are formal languages, and how many are linked to misperceptions of the human/computer dialogue. And, as has already been pointed out, what happens when these two factors come together in a language which has both a 'programming' and a 'logical' interpretation, as is the case with Prolog?
To begin answering these questions it is crucial to examine what happens when beginners start to construe the programming task, and to pinpoint two issues. Firstly, the way in which novices themselves construe or interpret expressions in the programming language needs to be explored to establish whether or not in fact they do interpret them as a form of natural rather than formal language. Secondly, it is important to investigate the way in which novices interpret the machine's behaviour to see whether the types of explanation of events they offer are consistent with some 'appropriate' view of the machine, or whether their explanations are over-reliant on the attribution of more intelligence or intentionality to the computer than is warranted.

These issues have not been fully addressed in the study of novice programmers and yet they are very important factors in the complex milieu of learning to program. Investigation is problematic since it is hard to devise foolproof ways of pinning down students' interpretations and underlying tacit assumptions. To make this problem slightly more tractable, this chapter exhibits a framework which represents three levels of discourse associated with Prolog programming (general discourse, logical discourse and mechanistic discourse). Each level of discourse is broken down into components briefly outlined in the previous chapter and further elaborated below. The main function of the framework is to represent natural and formal discourses and their differences, based on the discussion of formal reasoning in Chapter 3, and to organise a space of possible high-level assumptions and/or misconceptions which students have about Prolog.

The contention is that a great deal can be learnt about novices'
perceptions of programming and the computer by observing the language they use to describe their programs, and by noting the types of explanations they offer for their reasoning. These provide clues as to which discourse level the student is working at, and these can be contrasted with what they should be doing in an ideal case. Mixed (or muddled) discourse explanations indicate confusion or lack of knowledge and may therefore also indicate potential error. The framework also provides the structure and background for empirical observational studies to be reported in later chapters.

We begin by identifying and clarifying the notions of levels of discourse and discourse components.

4.2 Levels and Components of Discourse

Human discourse is multi-layered in the sense that people often set up specific discourse rules in particular situations which operate in the context of more general and more deeply rooted discourse understanding processes. These specialised discourses differ significantly from ordinary discourse, and yet the ways in which they differ from it, and from each other, may not be immediately obvious to a beginner, a point which is frequently overlooked in studies of novice programmers. The type of learner we have in mind is the beginner who comes to programming with general skills and knowledge about the level of general discourse, and practical problem solving skills, but no specialised knowledge about formal problem solving using either logic or programming (e.g. undergraduates and graduates).

Since our main focus is ultimately on the activity of programming, the analysis and discussion is confined to those types of problems
which can be solved by producing either a complete specification of a task, or a set of instructions sufficient to achieve a given goal. So we begin the analysis with the sub-domain of general, ordinary discourse associated with problem solving.

In the diagram below, the two major discourses are general and formal. For the purposes of subsequent discussion, practical problem solving will be regarded as part of general discourse, whereas formal problem solving is not. In the diagram below this distinction is represented by the use of a double dashed line. The discussion in Chapter 3 focused mainly on issues associated with understanding the consequences of moving between practical and formal problem solving.

![Diagram](image)

**Fig. 4.1 Discourses for Prolog**

But because Prolog programs (at least in theory) can be interpreted either as logical specifications or as more conventional, instructional programs, two formal discourses are required: the logical and the mechanistic. Both discourses are comparatively high level descriptions of programs. In the later part of the discussion it becomes important to determine which of the two formal domains a particular student is trying to use and understand.

Because we are interested in novices, we assume that problem
solving activities normally take place within ordinary discourse in
the real world, and putative solutions (or partial solutions) are
mapped into the terms of the formal discourses, a process which
will be discussed in more detail later in this chapter. This
mapping is liable to be a fairly explicit and laboured process and
somewhat error-prone in the beginning. As expertise develops so
the learner will be able to move into a formal domain and, being
aware of its structure and components, be able to generate
appropriate candidate solutions, no longer finding the need to move
back and forth in so cumbersome a fashion.

A particular difficulty is that the student must use everyday
English to communicate about formal notions: we do not talk or
think in logic or Prolog. The learner must eventually come to
understand formal problem solving in general, and the specialised
formal discourses in particular, through general discourse
processes. It is no wonder that some of these more general
processes find their way into the formal domains, as was
illustrated in the literature on deductive reasoning in the
previous chapter. Carrying assumptions from the real world and
ordinary human discourse into formal reasoning can also allow the
problem solver jump to conclusions, a process which is
characteristic of human problem solving, and which can often be
crucial to effective performance.

McCarthy (1980) has described 'circumscription' as a rule of
conjecture which can be used to allow jumping to conclusions about
situations described in formal problem solving. For example, in the
Missionaries and Cannibals problem, we normally would suppose that
the boat is useable unless we could deduce otherwise from the
problem statement:

'[Circumscription] will allow us to conjecture that no relevant objects exist in certain categories except those whose existence follows from the statement of the problem and common sense knowledge. When we circumscribe the first order logic statement of the problem together with the common sense facts about boats etc., we will be able to conclude that there is no bridge or helicopter.' (p28)

This is an interesting and useful way of formalising 'common knowledge' for the purpose of enabling computer programs to behave more effectively in problem solving situations. However, when people interpret problems, the opportunities to make assumptions are many and varied. For example, in the Missionaries and Cannibals problem, common sense might dictate that we assume the boat can be rowed across the river (i.e. it is not only waterproof, but also has oars), but equally, the same common sense may dictate that cannibals cannot be trusted to cross back and forth across the river unaccompanied by missionaries. Therefore, the range of possible moves is restricted in some peoples' interpretations which may mean that they will fail to solve the problem.

We have already examined abduction as a means of jumping to conclusions and noted that a strong contributory factor in abduction is the confirmatory bias to be found in most human reasoning. That is, if the reasoner decides that something is the case, then confirmatory evidence will be sought to confirm this. Thus, in the above case, if the worthiness of the boat is to be assumed on the grounds that we are not told otherwise, then equally, the lack of reference to the co-operativeness of the cannibals may be taken as confirming evidence that they may not be left unattended. It would seem therefore that people solving problems make assumptions based on common-sense ideas at various
points during problem solving.

To try to pin these down, we identify the sorts of assumptions associated with the four components identified above. Assumptions may be made whilst

(a) interpreting the language of the problem statement (i.e. using natural language understanding in an unconstrained way);

(b) reasoning about what objects and relations can be identified in the problem domain (i.e. as in the cases above);

(c) providing a solution which in fact is formulated to be read by another human, rather than any other abstract computational system (i.e. one which needs co-operative rather than analytic comprehension);

(d) evaluating progress, or success so far, in accordance with pragmatic considerations, many of which may be confirmatory in nature, blinding the problem solver to certain types of inadequacies in the putative solution.

This list is not exhaustive but represents the main points in formual problem solving at which novices may be thrown into conflict with natural discourse because the kinds of assumptions made during these phases for natural discourse and formal discourse are often quite different.

The four main components of ordinary discourse with which these phases correspond are those identified at the beginning of Chapter 3, as follows:

Component 1: the interpretation of language;

Component 2: understanding constraints on the discourse and what can or cannot be assumed

Component 3: the use of language (generation);

Component 4: the interpretation of behaviour, and the reconciliation of actions with the content of the interaction so far.

Fig. 4.2 Components of Discourse Understanding
The specialised formal discourses relevant to Prolog programming (the logical and the mechanistic) are broken down into the four components as follows. Component 1 (language interpretation) will be regarded as understanding the 'input' to a problem solving episode. That is to say, the input to problem solving may be a problem statement expressed in natural language, or a set of formal expressions. Component 2 (understanding the contraints on the discourse) is taken to mean being aware of the kinds of reasoning strategies which are appropriate for a given domain of discourse. This would involve understanding the general rules that govern formal reasoning (what to do) and the types of reasoning which are appropriate for a specific domain (how to do it). Component 3 (language generation) is regarded as the 'output' of a problem solving episode. Component 4 represents an evaluation process, which will need to take into account the sorts of changes which have been effected in the problem domain by actions resulting from expressions/instructions. The evaluation phase, therefore, does not refer to the assessment of the student's output by a tutor or experimenter. It is rather a reflective evaluation by the problem solver herself, in which she might try to imagine, or mentally simulate, what interpretations of her output could be made by an 'other'. The evaluation process may dictate changes in either the original form of the solution, or in its form of expression.

<table>
<thead>
<tr>
<th>Components of discourse---------&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
</tr>
<tr>
<td>INTERPRETATION</td>
</tr>
<tr>
<td>REASONING</td>
</tr>
<tr>
<td>OUTPUT</td>
</tr>
<tr>
<td>REFLECTIVE STRATEGIES</td>
</tr>
<tr>
<td>EVALUATION</td>
</tr>
</tbody>
</table>

Fig. 4.3 Four Components of Discourse

Discourse understanding is a cyclic process, and the above
components are meant to represent phases in this cycle. It is not the cycle itself which causes problems - formal discourses can also be described by the above components. Rather it is the assumptions which are often made at different stages in the discourse which lead reasoners astray. The components are closely interrelated and the problem solver would need to jump back and forth between the components to arrive eventually at an understanding of the problem to be solved and a possible solution. Therefore, the problem solver may not fully complete each phase of the problem solving cycle before moving on to the next, proceeding on the basis of partial understandings, or with an as yet incomplete model.

4.3 Combining Levels and Components into a Framework

Putting together the three discourses levels and their components, the basic structure of the framework emerges as follows:

<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Reasoning</td>
</tr>
<tr>
<td>INTERP.</td>
<td>STRATEGIES</td>
</tr>
<tr>
<td>General</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td></td>
</tr>
<tr>
<td>Mechanistic</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4.4 The Structure of the Discourse Framework

An important issue which needs to be highlighted is that the main distinction between ordinary and formal discourses has to do with
the degree of explicitness required both in terms of expression and strategy. To illustrate: understanding language in general discourse would normally involve some amount of reasoning. But, as has already been pointed out in Chapter 3, it is precisely this kind of reasoning that is taken for granted by learners - it is implicit, or tacit, reasoning. On the other hand, reasoning about how to specify the steps in a problem solution is usually recognised as an explicit activity. This distinction between 'implicit' reasoning, and 'explicit' reasoning reflects similarities to Johnson-Laird's (1983) notion of implicit and explicit inference processes in deductive reasoning described in Chapter 3.

Ordinary discourse, therefore, is one in which implicitness is commonplace because it is primarily the discourse in which we expect to be interacting with other people, who can use their own interpretative powers to fill in unstated details in expressions or sequences of actions. That is to say, in ordinary human discourse, forms of expression can contain implicit information to be derived or provided by the listener as part of discourse comprehension. However, expressions in formal domains should explicitly represent some state of affairs. The only implicit information permitted is that which can be derived using principled methods (e.g. deduction). With regard to strategy, intuitions or guesses which lead to a solution may be permissible in the general level of discourse, but such leaps must at some stage be substituted by the appropriate working out of the move in accordance with the constraints upon the formal domain. It is this process which beginners frequently fail to accomplish, resulting in a situation
where the reasoner has an answer, but does not know why it is correct. Just as reasoners are often confused because they think that implicit inferencing is adequate in deductive reasoning tasks, so it seems that students are confounded by implicit problem solving processes, which are either mistaken for explicit ones, or are not recognised at all. This results in students producing specifications or programs which still contain implicit information because they have not been transformed into the explicit steps required at the formal levels.

The framework will ultimately be used to represent what a student 'really' did during problem solving/program writing as opposed to what the student believed herself to be doing. It will be possible to illustrate in more detail not only how general discourse processes interfere with problem solving, but also how they constitute effective learning strategies. We described abduction as a powerful technique when it happens to be correct, but potentially disastrous when incorrect: these two faces of the same coin can be dubbed 'superstrategy' versus 'superbug'. The use of human discourse strategies are similarly well described as superstrategies and superbugs.

We now clarify and fill in the components of discourse at each level in the framework.

Input Interpretation

This column represents the way in which the input to the problem-solving cycle is interpreted. Interpretation of input at any level will be a two-phase process: firstly to understand what the 'symbols' in the problem statement are, and secondly to derive from
them a problem domain or model of the task.

For example, if the input is in natural language the problem solver must interpret the problem statement as a piece of text - i.e. by making all the necessary implicit inferences which are part of understanding ordinary language. Secondly, the problem solver must explicitly work out what the task is supposed to be. This will involve identifying the objects and relations in the problem statement. As we have seen, this process can be surprisingly difficult, and is sometimes adversely affected by linguistic features of the presentation. If this phase goes badly then the problem-solver may find herself in difficulties because she is inadvertently solving a different problem from the one described in the problem statement. Alternatively, she may fail to produce a model at all (e.g. as may be the case with the Himalayan Tea Ceremony).

These two activities correspond to Hayes and Simon's (1974) 'language process' and the eventual construction of a 'problem space'. The construction process in this part of the discourse framework is envisaged as producing only a basic correspondence model which will require further refinement when constraints on the reasoning methods are identified (i.e. moving to Component 2: reasoning strategies).

The cells are labelled 'implicit' for natural language, and 'explicit' for the two formal domains (see Figure 4.5).
A problem inherent in this phase is mistaking the interpretation of language (using implicit inference) for understanding the task sufficiently well to solve it, when in fact all that has taken place is a process of 'making sense' of the problem description at the general level. In this case, distinctions between implicit comprehension processes and explicit reasoning steps are blurred. Implicit assumptions may be made as part of language understanding which then become part and parcel of the correspondence model, but which are not explicitly identified. These assumptions can govern the success or otherwise with which the solution is achieved. An example of this is as described above, when students assume that no cannibals can be trusted to cross the river on their own, since in the real world such a move is liable to result in disaster. This assumption may remain implicit and unquestioned for a long time before the problem solver realises that there is nothing in the formal problem domain to preclude that move - even though there is nothing to include it either! The implications are, therefore, that several different but perfectly reasonable interpretations could be
made of a problem situation from a natural language input, and only a few (or possibly only one) will result in a correct solution.

If the input to the problem solving episode is in the form of logic (e.g. some expressions in first order predicate calculus which are to be clausified) or if it is a piece of program, then the situation is somewhat more constrained. For example, Van Someren (1987) points out that novices frequently interpret programming problems as instructions to devise a method for performing a task which they would use, taking no account of the fact that the computer must use it too. He notes also that the form of presentation of the problem is crucial: if it is couched in the datastructures of the implementation language, then the problem stands a chance of being solved. If however it is expressed in natural language, beginners often fail to produce a running program because their interpretations are not constrained to the specific programming methodology associated with a given programming language.

In the terms of the framework, the situation described by Van Someren can be described as one where general discourse processes are preferentially used by novices (producing instructions which are appropriate at the general level, but not at the formal mechanistic level) unless there is something in the problem statement which cues them (or reminds them) to do otherwise. Most important is the fact that students fail to solve the problem when natural language is used in problem presentation because interpretation is too free.

It is worth pointing out, however, that simply asking untutored
students to be more explicit than ordinary discourse understanding requires raises the question 'how explicit?' (Pea, 1986, p.34) - i.e. what can be taken for granted, if anything?

This difficulty arises in both the formal domains, though the consequences are different for each. In logic, as we have seen, problems arise because untutored people assume that logical expressions can be interpreted in the same way as ordinary language. The problem here is that many inference processes are implicit in ordinary language understanding - i.e. the untutored subject may simply not be aware of the number of inferences made in normal discourse. Therefore, students can believe that what they have written could not be made more clear when in fact it is evident from the formal perspective that some expressions are imbued with non-existent nuances of meaning (for example, assuming that the logical 'and' has the meaning 'and so').

Programming languages, on the other hand, often do 'embody' or implicitly represent certain kinds of information - for example activities which do not have to be explicitly stated (e.g. the unification process in Prolog which takes place automatically). This problem is particularly acute in the case of Prolog because of the automatic mechanisms. For example, a study which will be reported later in the thesis involves students interpreting a Prolog program to describe its backtracking behaviour. One of the most difficult facets of this task is keeping track of what the current variable instantiations are, since variables are bound by forward execution and unbound by backtracking. None of this complex behaviour is contained explicitly in the text of the program. Beginners must learn to hallucinate such activities onto the code,
and, as we shall see, they adopt many strategies for coping with this, only some of which are effective.

Reasoning Strategies

This column represents the 'problem space' derived from the input interpretation process. The problem space for each discourse has different characteristics. Practical reasoning can make reference to various knowledge sources, whilst formal reasoning is normally confined to that given in the problem statement, or the premises. An important difference between general discourse and the two formal discourses is that in general discourse, a premium may sometimes be upon simply obtaining a solution, by whatever means. In this case, guessing, intuitive insights, rememberances of previous experiences with problems of a certain type and so on may all contribute. However, in the formal domains, it is essential to use expressions to explicitly represent either what needs to be true for the solution to be true, or alternatively, how to accomplish the given goal.

These two aspects are designated by the label 'analytic', and are contrasted with the label 'ordinary' (see Figure 4.6).
<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse----&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>INPUT</td>
</tr>
<tr>
<td>GENERAL</td>
<td>Implicit</td>
</tr>
<tr>
<td>---&gt;</td>
<td>(Natural)</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>Explicit</td>
</tr>
<tr>
<td>---&gt;</td>
<td>(Logical)</td>
</tr>
<tr>
<td>MECHANISTIC</td>
<td>Explicit</td>
</tr>
<tr>
<td>---&gt;</td>
<td>(Programming)</td>
</tr>
</tbody>
</table>

Fig. 4.6 Reasoning Strategies in the Discourse Framework

Strategies from the general level may contribute indirectly to problem space construction at the lower levels. That is, whilst considering how to solve a puzzle in formal way, people do not restrict themselves only to thinking 'formally' (even if that were possible!). They consider how they could possibly solve the problem from the practical viewpoint, taking into account the formal restrictions, so that, eventually, the problem solution is valid from the formal perspective. That is to say, a formal domain can only be used to represent certain types of reasoning (e.g. deductive). So even if initially the problem solver finds a solution by intuitive, practical means (e.g. by visual cueing, or guessing), she must find a way (if there is one) of expressing the solution within the confines of the formal system - a lucky guess cannot be legally represented in logic, or in a program, even if the answer could. Such a guess must be followed up by principled substitution.
Output

Having established a solution, the problem solver then must generate some output. The term 'output' is to be interpreted in a liberal sense to mean any statement which is a result of the reasoning process. This may include a verbal statement, a diagram, a specification couched in one of the discourse languages and so on.

This column reflects the input language column to some extent, in that the forms of expression appropriate to each discourse are again designated implicit or explicit, although the terms have been generalised so as to include the possibility of diagrammatic representations. The sense in which diagrams can still be regarded as domain specific is that the problem solver may produce a diagram of the problem situation (as in a correspondence model), a diagram of the logical specification (e.g. AND/OR trees) or a diagram aimed at representing the mechanisms which are to be implemented in a program (e.g. a flowchart).

<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td>V INPUT</td>
<td>REASONING</td>
</tr>
<tr>
<td>INTERP.</td>
<td>STRATEGIES</td>
</tr>
<tr>
<td>GENERAL</td>
<td>Implicit</td>
</tr>
<tr>
<td></td>
<td>(Natural)</td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td>(Logical)</td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
</tr>
<tr>
<td>MECHANISTIC</td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td>(Programming)</td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td>Language</td>
</tr>
</tbody>
</table>

Fig. 4.7 Output in the Discourse Framework
Reflective Evaluation

At the general level, one simple way to check that a solution (or set of instructions to accomplish some task) is not only sufficient, but correct, is to give the instructions to another person to perform and to see if (or how well) they achieve the goal. This activity is often transmuted into an internal mental simulation, where the problem solver herself attempts to re-interpret instructions as if she were another person. It has already been noted in relation to mental model generation that the 'de-centring' that is required for mental simulation can present difficulties, due to the inclination toward cognitive set, and confirmatory bias.

In the mechanistic domain, a common evaluation technique is to run the program and to judge the computer's behaviour as appropriate or otherwise. This technique has a long history in the fields of Computing and Artificial Intelligence. Because mental simulation of programs of any size is difficult, direct observation of behaviour, when available, is essential.

But on this issue, the two formal domains differ markedly because expressions in logic are not normally regarded as instructions to act, but are interpreted according to model theory. Alternatively, the manipulations of logical expressions may be treated as purely syntactic. Within the domain of logic, as has been pointed out in Chapter 3, it can be difficult for beginners to be generate sufficient models to be certain that their conclusions are correct.

In the programming domain, logical expressions may need to be interpreted as instructions to act, but in order to do this they
will be taken from the purely logical domain and re-interpreted in another domain: i.e. the real world or the mechanistic. This is, in effect, the extra level of interpretation discussed in the previous chapter with regard to declarative interpretations of Prolog - i.e. they are often not directly linked to actions. If they are to be interpreted as actions, then they depend on another set of rules to perform this transformation, hence the extra level of interpretation. Effective reflective evaluation may therefore consist of a process as complex as establishing that in all possible worlds a given logical specification holds true (i.e. theoretical) right through to the simple question on many novice programmers' lips: 'Does it work?' accompanied by running the program (i.e. empirical).

There is a significant difference between the theoretical and the empirical methods. As was pointed out in the previous chapter, hypothetical validation is appropriate to formal reasoning methods (i.e. proof according to some theory), whilst empirical validation is appropriate for practical reasoning (i.e. statements can be demonstrated to be true by reference to known facts or observations of events). The cells in the framework are labelled accordingly (see Figure 4.8).
The computational domain has similarities both with the formal logical domain and the general domain. That is to say, the empirical method of observing behaviour is appropriate for certain purposes, but equally, the programmer may use hypothetical evaluation methods, particularly as programming skills become more sophisticated.

Having constructed the framework, some further points need clarification. Studies of novice programmers have tended to be intra-discourse studies at the mechanistic level of discourse. That is, the focus of attention has been on how learners relate the components of mechanistic discourse to one another: e.g. how to represent particular types of reasoning strategies in the expressions of the language, and how to relate the machine's behaviour to expressions which the learner has produced in a program. The issue of inter-domain interference has not been ignored, but has tended to be underemphasised.

Similarly, in the studies of deductive reasoning, the output of
non-logicians is often evaluated as either correct or incorrect according to the rules governing logic: i.e. the relationship of the input to the output of the subject is compared and evaluated at the level of logic.

In the next section, the various ways in which learners can move around the framework are described. Then, to illustrate how the framework can be used to clarify errors the general and an abstract formal level will be used to represent some of the errors in deductive reasoning and to re-evaluate some of the studies in programming reported in the previous chapter.

4.4. Moving Round the Framework and Representing Misconceptions

There are three ways of moving around the framework: horizontal, vertical and diagonal. Horizontal moves represent problem solving viewed from within one discourse: i.e. moving across the framework within a discourse is to describe phases of problem solving at that level.

Vertical moves represent mapping one component of discourse into another discourse. In order for these types of moves to be successful, the problem solver needs to be aware of principled ways to accomplish them. These principled ways are represented as straight lines. A vertical line involves a systematic re-interpretation: e.g. the line from general discourse output to either of the formal discourses is one along which expressions must be made explicit. The line between logical expressions and programming expressions is one on which the underlying model for interpretation must change from logic to mechanistic semantics.

Horizontal lines between components in a discourse also represent
consistent and correct mappings between, say, the interpretation of a problem statement, through the reasoning process to the output and evaluation.

Note that for the purpose of the discussion in this section, the logical and mechanistic levels have been collapsed together into simply a 'formal' domain.

The framework has a direction and can be wrapped around - the student is envisaged as progressing from left to right and looping back into the framework. For example, given the Missionaries and Cannibals problem, a student may interpret it (Interpretation of Input), producing an initial correspondence model such as that in Figure 4.9. The student may reason that this representation is not what she wants (Reasoning) because she is

```
State 1: (ccc, mm, b) --- (0,0,0)
State 2: (c, m, 0) --- (c,m,b) etc...
```

where the left most parentheses designate the left bank, and the rightmost parentheses the right bank. There is no suggestion in this example that this is a necessary or an optimal form of representation, only that it is a reasonable candidate amongst many possibilities. Also, reasons why a subject chose this particular
representation for the problem, and no others, will in general not be known. In other words, we are not interested in the entire space of possibilities which is available to the novice. Having decided on this she must evaluate whether this is a good representation. This in turn may involve interpreting it, reasoning about it, drawing a conclusion and evaluating that - and so on. As described, most of this activity is taking place at the general level: these are meta-level decisions which are not necessarily particular to any formal discourse. Certainly, the decision to change representation on the basis of using a typewriter (where the diagrammatic representation is more difficult to type) is a general level issue. Had the decision been motivated by a constraint imposed by a formal domain (e.g. logic does not normally use pictorial representations) then a mapping would be in the process of being constructed between the general and the logical discourses.

Progress around the domains of discourse can be represented as 'square' moves. This is illustrated on a simplified framework diagram which comprises only the general and a formal discourse. In this particular case, the problem solver has been provided with a natural language input and is required to generate a formal solution as output (see Figure 4.11).
The pattern in Figure 4.11 represents the final track left at the end of the problem solving episode. That is, several cycles of implicit problem solving may have taken place before this final route was established. Because only straight lines are shown, this representation illustrates that the problem solver has successfully mapped between the two domains (hence the vertical lines) and has mapped between components of problem solving at both levels (hence the horizontal lines). The final exit track is also at the appropriate level for the problem.

What happens when things go wrong? 'Wavy' lines in the framework represent implicit or intuitive moves, and although such moves may play a crucial role in deriving an eventual principled solution, wavy lines need to be replaced by straight lines to be effective from the formal point of view. Unless this substitution takes place, the end product is liable to be incomplete from a formal perspective because assumptions made as part of an intuitive move will have been left implicit in the solution. This incompleteness may be found at two levels: (a) the solution itself is incomplete - i.e. steps are missing; or (b) the way in which the solution is expressed leaves information implicit to be derived by 'common

---

<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INPUT</td>
</tr>
<tr>
<td></td>
<td>REASONING</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
</tr>
<tr>
<td></td>
<td>REFLECTIVE</td>
</tr>
<tr>
<td></td>
<td>INTERP.</td>
</tr>
<tr>
<td></td>
<td>STRATEGIES</td>
</tr>
<tr>
<td></td>
<td>EVALUATION</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GENERAL</th>
<th>Implicit</th>
<th>Ordinary</th>
<th>Implicit</th>
<th>Empirical</th>
</tr>
</thead>
<tbody>
<tr>
<td>*---&gt;</td>
<td>*---</td>
<td>*---</td>
<td>*---</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FORMAL</th>
<th>Explicit</th>
<th>Analytic</th>
<th>Explicit</th>
<th>Hypothetical</th>
</tr>
</thead>
<tbody>
<tr>
<td>*---</td>
<td>*---</td>
<td>*---</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Fig. 4.11 Ideal movement patterns round the framework
sense'.

To illustrate, the pathway in Figure 4.12 represents what some untutored subjects appear to do in some deductive reasoning tasks:

<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>REASONING</td>
</tr>
<tr>
<td>V</td>
<td>INTERP.</td>
</tr>
<tr>
<td>1: GENERAL</td>
<td>Implicit</td>
</tr>
<tr>
<td></td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>2: FORMAL</td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td>$\rightarrow$</td>
</tr>
</tbody>
</table>

Fig. 4.12 Performance in Deductive Reasoning

In this case, where the input was some logical expression, the problem solver has moved directly to general level discourse of input interpretation via a wavy line, and she has stayed within the general level throughout the rest of the problem solving episode.

In other words, this subject has:

(a) interpreted the input as if it were natural language (Erickson, 1974; Evans 1982; Johnson-Laird and Wason, 1977; Henle, 1978; Chapman and Chapman, 1959) illustrated by the wavy line from formal input to general input interpretation;

(b) used ordinary, informal, practical reasoning skills (Woodworth and Sells, 1935; Begg and Denny, 1969) - the line of progression through problem solving remains at the general level;

(c) constructed a solution which is not a formal solution (a reflection of (a) above) - i.e. the output solution is implicitly rather than explicitly formulated;

(d) and has evaluated that solution using empirical techniques (Cole, 1977) - note that the exit point is from the general level of discourse, rather than at the appropriate formal level.

Note that Henle's (1978) position and the 'illicit conversion of premises' results (Chapman and Chapman, 1959) are associated with
interpreting input, rather than reasoning: the problem for subjects is not that they reason incorrectly, but that they begin by misinterpreting the givens.

Also the above representation implies that despite this initial 'incorrect' move to the general level of discourse, the subject has been able to successfully map from one component to another at the general level. This in turn implies that the problem solution is correct in some sense, but not necessarily at the 'right' or appropriate level of discourse.

There are several points of discussion arising from the pathway depicted above. The first is that it is the one we would expect complete novices to follow, because they have as yet no knowledge of the underlying formal discourse of logic: they cannot map into some discourse of which they are unaware. Looking back to Miller's (1981) experiment, for example, the behaviour of his subjects could similarly be represented.

Recall that in this experiment, non-programmers were asked to provide natural language instructions for 'someone else' to perform a card indexing task. Miller's subjects, naturally enough, believed themselves to be writing instructions in the real world (i.e. for another person to interpret) since there were no instructions to do otherwise. Using practical reasoning strategies, and using English, which allows information to remain implicit, they wrote instructions which they thought they (or another person) could interpret sensibly to perform the task. Miller then proceeded to analyse those instructions to see if they would be adequate at a formal level. His subjects followed the pathway illustrated in
Figure 4.13.

<table>
<thead>
<tr>
<th>Discourse</th>
<th>Components of discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERAL</td>
<td>Implicit</td>
</tr>
<tr>
<td>FORMAL</td>
<td>Explicit</td>
</tr>
</tbody>
</table>

Fig. 4.13 Pathway used by Miller's Subjects

This pathway illustrates that, as far as the subjects were concerned, the formal level simply did not exist. They were non-programmers, and nothing in the problem presentation indicated that the assumptions they made were not correct. Furthermore, the task instructions were in English. It is unsurprising, therefore, that specifications for the task were incomplete, though it would be interesting to see how far 'missing' information could be supplied by ordinary inferences of the sort normally used in natural language understanding (i.e. a problem with output). Large gaps in the instructions might indicate that individuals had failed to de-centre sufficiently well to imagine how the instructions might sound to a newcomer to the task (i.e. a problem with evaluation). That is, the poor performance which Miller reports need not be conceptual or expressional, implying that the subjects could not do this task, but may be due to a higher level assumption by the participants - that the anonymous interpreter would be as intelligent and well-informed as themselves.

This assumption is a reasonable one, given the discourse level
within which subjects were working. The tendency to assume common sense on the part of the recipient of instructions has been noted in the programming literature, and it inevitably gives rise to incompleteness of specification. Bonar and Soloway (1985) specifically designate plans for programs which are in fact instructions for humans to perform a task (which is what Miller asked his subjects to produce) as 'natural plans', and note that it is often difficult to see what a plan means at the mechanistic level because a great deal of information is left implicit or ambiguous.

If we represent Miller's task on the framework, we can see that the trackway he expected of his subjects is complex, involving mappings back and forth between domains (see Figure 4.14).

Figure 4.14 illustrates an ideal pathway, but if subjects were unaware of the hidden formal aspects of the task, they could not possibly perform these mappings because the appropriate domain knowledge did not exist. These studies, therefore, illustrate subjects working without the benefit of required information, and that they managed to produce solutions which were in any way
sensible is a credit to the subjects.

Importantly, however, as students learn, they will need to start constructing an understanding of the formal levels, but this development is necessarily mediated by general discourse processes. During problem solving, meta-level reasoning and decision making will take place at the general level of discourse. For example, if students are presented with logical expressions, or programs as input to problem solving, they will need to reason about what these hieroglyphics 'mean' to them before they can produce appropriate responses at the formal levels. This understanding process will take place at the general discourse level, and may involve both implicit and explicit reasoning.

Of course a major problem for this view of problem solving is how to distinguish between intuitive and principled methods of moving around the framework when observing a student working. This will remain a difficult issue throughout this discussion, and subsequently when protocols are examined in later chapters. Nevertheless, given that problem solving has been divided into various discourses, it does seem that the language used by a student to describe his or her activities, and the sorts of justifications which are offered for certain types of move in the problem solving process, can be classified as appropriate to a given discourse. This in turn can provide crucial information about the discourse within which the student is currently operating. That is to say, in idealised logical problem solving the student might locate herself within logical discourse, and use general discourse as a referent: therefore, in generating a model for interpreting some logical expressions, she can select out from the range of
available real-world models only those which fit the constraints imposed by the logical domain. On the other hand, the novice problem solver is liable to be located within the general discourse level, and may construct putative models of states of affairs which she then attempts to map onto logical expressions by trial and error. Quizzing the first student with regard to decisions about models is likely to elicit statements about the constraints of the logical domain (e.g. 'I can't do that because this expression precludes such an assumption'). Quizzing the second student is more likely to elicit information about the real-world (e.g. as in the above example, captive cannibals cannot be trusted).

To illustrate, assume that a student has been presented with a set of logical expressions from which she must derive some information. The starting point for 'input' would be therefore be at the logical level. In order to begin interpreting what such expressions mean, she will need to move upwards to the general level. She may go directly to the reasoning component rather than via input interpretation at first the logical and then the general levels. Moving into the general domain might feasibly involve substituting concrete objects for variables in logical expressions. The product of this reasoning is output at the general level: i.e. a statement, perhaps, of what she thinks the logical expressions mean, which should be evaluated. This would involve re-entering the problem solving cycle with the output as new input, but this time coming in at the general level, rather than the logical.

What the problem solver must understand at this point is that at some point she will need to move back from the general to the formal domain. The method of her derivation for new information
must eventually be made explicit in the terms of the logical discourse in order that the solution can be regarded as sufficient and correct from the formal point of view. This may entail re-instating variables where concrete objects or facts had been substituted and so on. The discourse framework can be used to illustrate the eventual pathway in Figure 4.15.

<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INPUT</td>
</tr>
<tr>
<td></td>
<td>V INTERP.</td>
</tr>
<tr>
<td>GENERAL</td>
<td>Implicit</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>FORMAL</td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4.15 An ideal pathway for formal problem solving

However, several things might go wrong. The most straightforward would be that the novice simply forgets that this last step must be performed. So, she may have a perfectly good solution, but it is expressed in terms of the general level of discourse. The pattern therefore might be represented as in Figure 4.16.
<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discourse</td>
<td>-------------------------</td>
</tr>
<tr>
<td>INPUT</td>
<td>REASONING</td>
</tr>
<tr>
<td>V INTERP.</td>
<td>STRATEGIES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GENERAL</th>
<th>Implicit</th>
<th>Ordinary</th>
<th>Implicit</th>
<th>Empirical</th>
</tr>
</thead>
<tbody>
<tr>
<td>---&gt;</td>
<td></td>
<td>---&gt;</td>
<td>---&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FORMAL</th>
<th>Explicit</th>
<th>Analytic</th>
<th>Explicit</th>
<th>Hypothetical</th>
</tr>
</thead>
<tbody>
<tr>
<td>---&gt;</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4.16 Pathway indicating a 'slip'

There is one wavy line from formal input to general reasoning strategies, and from formal reasoning to general output. The first is in effect 'cancelled' by the vertical (hence principled) mapping from general reasoning to analytic reasoning. The second remains uncanceled. However, in this case, it should be relatively straightforward to correct the error, and re-represent the solution in formal terms because the reasoning strategies were mapped into the formal domain, so the constraints on reasoning imposed by the formal domain should have been taken into account. Asking this student to re-represent the solution should not produce great cognitive confusion though there may be local difficulties in, say, using the symbols of logic to correctly capture what she means.

Alternatively, it may be that in moving upwards to the general level in the initial phase of problem solving, the student began to perceive the problem as a 'real-world' problem, and generated a solution which could no longer be easily transformed back into logic. The representation is therefore of a situation in which steps in the problem solution are liable to be omitted - a much more problematic case than that described above.
On the other hand, Figure 4.17 illustrates the case where the only wavy line (from formal interpretation to general reasoning strategies) is uncancelled, indicating that the student has not mapped back into the formal discourse at any point, and it is therefore possible that it will not be easy to render her solution into logic.

These two examples highlight the importance of the reasoning phase in problem solving. Because of the important role of reasoning, a pathway which we would not expect to see is that in Figure 4.18 where the student has jumped up from formal input interpretation to ordinary reasoning, from there to explicit formal output, and then back to general reflective evaluation.
The particular jump which is problematic is that from ordinary reasoning to explicit representation - the product of ordinary reasoning cannot be output which is formally explicit in one jump. The problem solver would need to go through a phase of producing implicit output which is then transformed into explicit formal output by a principled method. Even if the problem solver believed herself to be producing explicit output, because the reasoning process was at the general level of discourse, the constraints of the formal domain will not have been taken into account. Therefore, the solution is likely to contain implicit information still.

Students who have not recognised the formal nature of the domain may solve set problems for themselves by practical means and when trying to transform this practical solution into formal terms (either of logic or of the programming language) they may find that it either does not work, or that it involves a lot of local 'patchwork' because it is (in this context) 'ill-conceived', or a square peg in a round hole. The novice may have a great deal of difficulty in understanding what is wrong with her specification or program in this case.
These situations could have arisen for one of two reasons: either because the student unintentionally slipped from one domain to the other whilst working; or she may have remained at the general level because she was simply unaware of the formal level. The first case of simply forgetting to move back implies that the student was aware of the formal level, and her slip was to omit a mapping. The second more problematic case implies that the student was in fact unaware of the formal level in any meaningful way. The only method to establish which pathway reflects more about the true situation is to probe the problem solver's perspective and reasoning strategies to establish which discourse was used as her 'base' and which as the referent. Hence the need to talk to students about what they have done, and to identify what kinds of underlying perceptions are revealed by their justifications and rationalisations.

4.5 Superbugs and Bugs

The discourse framework can be used to illuminate the concept of superbug introduced by Pea (1986). Wavy lines represent superbugs unless these are replaced by straight lines. Bugs are defined as mistaken mappings either vertically or laterally in the framework. Superbugs, therefore, are linked with either ignorance of the formal domain or a misapprehension about the relationship of the formal domain to ordinary discourse.

The similarities between the general discourse level and the mechanistic (reflected in the framework) can easily give rise to difficulties for students. In introductory programming courses, these two discourses tend to be presented as similar to one
another. For example, as was pointed out in Chapter 3, it is possible to reflect in the computational domain the sorts of causal and temporal reasoning used to help solve problems in the real world from correspondence models. Also, Prolog can be made to look like English, and there seem to be certain similarities in the process of observing machine behaviour for evaluation purposes, and observing a human performing a given task according to instructions. Unfortunately, however, this similarity between discourses is only superficial, and cannot be maintained for long. It can therefore be a major source of difficulty for novices if they are introduced to programming via the apparently 'soft', or easy, route of identifying the similarities between the discourses, without due attention being paid to their significant differences.

What needs to be investigated is the way in which learners attempt to interpret the mechanistic level from the general level of discourse, and what typically goes wrong. This enterprise will eventually become programming language-specific, which is to say: the particular ways in which a given programming language presents itself (and is presented) to beginners will affect the sorts of assumptions they make, and will therefore determine the possible types of misapprehension to be expected. For this reason, although Pea (1986) has defined superbugs as 'language independent', the particular bugs which derive from a superbug will be language specific.

But before discussing Prolog-specific bugs, we first re-examine Pea's (1986) bugs briefly described in Chapter 2. Recall that the three bugs were:
* egocentrism bugs: students assume that the program is as meaningful to the computer as it is to them;

* intentionality bugs: students attribute to the program foresightedness or goal-directedness;

* parallelism bugs: students think that several lines in a program are 'active' or are known by the computer at the same time.

Whilst all of these bugs are examples of remaining within the general discourse level and pivot on the notion that computer is another human being, it is possible to link them to different components of the discourse framework. It is not possible to map them on the framework because Pea does not provide sufficient information about, for example, the type of problem statement given to his students nor their final output.

The egocentrism bug is a language bug, and, because it is linked with program writing (i.e. as far as the person is concerned, the program is her 'output'), it is associated with the Output component of the framework. The presupposition by the student is that the computer understands the language of the discourse as if it were natural language. Not only does this suggest that the computer should be able to make inferences to understand what the intentions of the programmer are (in a co-operative manner) but the further implication is that the program instructions provided by the student will be incomplete. This turns out to be the case:

> It is as if they do not see that the necessary specifications to the computer have been omitted. All they have provided is the skeleton of a program, assuming that in some way the computer can fill in the rest, can say what they 'mean'. (p.31)

Pea assumes that the difficulties are almost perceptual in nature, hence his emphasis on the word 'see' above. However from the foregoing discussion it should be clear that, because of the
unconscious inferences underlying natural language interpretation, there is no reason to suppose that students will spontaneously recognise that their specifications are incomplete.

Intentionality bugs and parallelism bugs are both evaluation bugs: students appear to use the empirical/behavioural evaluation methods appropriate to the general discourse level, rather than those appropriate to the mechanistic level.

This type of bug is compounded by the fact that, frequently, the most convenient language for describing the computer's complex activities is intentional language. For example, it is common to hear experienced programmers comment: 'it's trying to prove this clause', or 'it can't find an instantiation for that', or 'it's gone to look for more examples' and so on. It would clearly be cumbersome to always use proper mechanistic phrases to describe these kinds of activities - the intentional language is a convenient form of shorthand, which can be easily understood by a broad spectrum of people. But the novice must realise and acknowledge that this is an analogical use of language, a form of anthropomorphism. This must be borne in mind even though in many cases, intentional language is not only hard to avoid but also can be essential in helping the learner sidestep inevitable complexity of detail. Only by probing, questioning and observing the behaviour of the student can the distinction be made between those using intentional language as shorthand and those using it as a basis for explanation.
4.6 The Prolog Learner

Until now the discourse framework has been used in a quite general way to illustrate the sorts of difficulties formal domains present to beginners. Because learners use the real-world as a base for their reasoning they fail to recognise or make explicit the sorts of implicit assumptions which are commonplace in the real world of human discourse. We now consider the particular effects of this situation on Prolog learners.

So far we have used only two levels in the discourse framework at a time. Beginners would normally only try mapping from general discourse into one other formal discourse. But in the case of Prolog programming, it can be argued that all three domains are relevant: Prolog can be viewed as logic, or as a more conventional mechanistic programming language. This means that the two formal levels of discourse - the mechanistic and the logical - rather than being independent domains are related to one another. One discourse could in principle be used to explain solutions derived from another discourse: e.g. in the ideal situation, a logical solution could be explained by reference to the underlying mechanisms which implement it on a computer (i.e. by moving down to the mechanistic level), or could be explained by reference to hypothesised real-world objects or events (i.e. by moving up to the general level). However, these explanations are only likely to be correct when formulated by a user who already understands the relationships between the domains. Because the mappings between the domains are not always straightforward, such a user would already be in possession of the sophisticated skill of locating herself in any one of the domains, using any of the others as referents.
In order to capture the situation for Prolog learners, the logical level of the framework will be used to represent the logic based (or declarative) view of Prolog, and the mechanistic level the view of Prolog as programming language (the procedural interpretation). The following three sections examine (i) problems arising from the declarative interpretation, (ii) problems arising from the procedural interpretation and (iii) problems arising from a hybrid approach - i.e. interactions between the two formal domains.

4.7 Learning Prolog by a Declarative Approach

The declarative approach to teaching and learning Prolog would involve trying to emphasise programs as logical specifications with little regard for the execution strategy. Assuming that students have as yet no knowledge of logic, but that they do have general knowledge about problem solving in the real world, what can be used as a referent? A straightforward response to this question might be to teach students logic first, before teaching them Prolog. This would provide them with techniques for interpreting specifications taking into account the restrictions imposed by the use of Horn clauses, and the process of resolution (i.e. deriving a contradiction). This, in combination with knowledge of the underlying inference procedures to construct proof trees, would constitute a fairly robust basis for interpreting Prolog specifications.

Whilst this approach would in principle furnish the necessary requirements for a logical interpretation of Prolog, it constitutes a sideways move. We have already discussed the difficulties experienced by people trying to learn and use logic, and this
approach does not necessarily avoid any of them. In particular, a high premium is put on the novice’s own ability to generate representations, models and interpretations which she herself will evaluate. Given the confirmatory bias discussed in previous chapters, we would expect this to be an error-prone process.

Input Interpretation

In Column 1, therefore, students may interpret logical expressions as natural language. This throws into doubt the claim that one of Prolog’s advantages as a programming language is that it can be made to look like natural language. From our discussion so far it should be clear that there is no reason to suppose that this is any advantage at all, and in fact, may only serve to confuse and mislead.

This situation is compounded by the fact that Prolog does not have the same sort of syntactic markers common to other programming languages, with the result that so long as expressions have the general form of Horn clauses, syntactic restrictions are few. Because of this relative freedom of expression, problems of representation are rife, but frequently unrecognised by the unwary beginner. This issue was raised in Chapter 3 and suffice to say if expressions can be interpreted as natural language expressions, then some students will do so. It is likely that programs they create as output will also be underpinned by the semantics of natural language, and are liable, therefore, at the very least to be incomplete (whether or not they are correct).

For example, a description of the ‘brotherhood’ relation in English could take several forms: Henry and Peter are brothers; Peter and
Henry are brothers; Henry has a brother; Susan has a brother; the sons of the same parents are brothers; Uncles may be brothers and so on. The implicit information contained in these English phrases is that associated with gender, signified by appropriate names of individuals and the term 'brother' itself denotes a gender (in one direction). If provided with some Prolog clause defining brotherhood, therefore, the student using the general discourse level as a base has a certain amount of choice with regard to interpretation, but only a sub-set will be correct at the formal logical level. The notion that one can be one's own brother, for example, may not occur to such a learner and when Prolog offers this as a viable possibility, it may be discarded as silly or irrelevant.

The fact that Horn clauses are Prolog's logical basis introduces a further set of constraints associated with the particular way in which states-of-affairs can be represented at the logical level: Horn clauses have a restricted expressive power, and it follows, therefore, that some things which the problem solver might need to express, and which might be considered part and parcel of the problem solving domain, cannot be expressed in Horn clauses.

**Reasoning Strategies**

The difficulties in maintaining a logical view become more difficult as programs become more complex. For example, when any serious list processing is to undertaken, a purely declarative interpretation can not only be unwieldy, but also inadequate. Understanding a process such as recursive list processing declaratively is feasible if the tutor carefully selects examples,
but generating list processing clauses from first principles using a declarative interpretation would be difficult (c.f. creating for oneself the clauses defining the 'append' relation from a declarative perspective). The point is that several declarative versions could in principle be generated, but not all of them would work in Prolog. This type of difficulty may disappear with the advent of a 'real' logic programming language.

In Chapter 3 we discussed the difficulties for novices receiving a declarative introduction to Prolog, where declarations may not map easily into the sorts of correspondence models which result from interpreting problem statements (cf the River program used as an example in that chapter). It was suggested that procedural interpretations of programs can, at the very least, support the novice's first approximation of a problem solution constructed around a correspondence model. In this case, the interaction between reasoning strategies in the real-world based on procedures as actions and procedures at the computational level might be usefully exploited. However, the declarative approach to problem solving (typically problem decomposition) can often seem oblique to beginners.

Output

Basing program interpretation on natural language semantics will yield incomplete and perhaps badly structured programs. The problem is that a robust and consistent representation may not emerge from an informal natural language interpretation. Therefore, not only will programs contain implicit information, but they may also only 'make sense' in the real world and not at the level of logic.
Reflective Evaluation

Due to unawareness of the formal rigour underpinning logical specifications and expressions, it seems likely that beginners will use real-world, empirical validation techniques. In fact, this approach may be sufficient given that the computer will run the logical specification, but the problem is that the program may run for reasons which the novice neither acknowledges nor understands. The other side of this coin is that a specification can be logically correct but still not run. Finding out how to distinguish these cases with little or no reference to execution will be almost impossible.

It would seem that if students are taught only the declarative interpretation of Prolog, then most difficulties will focus around the interpretation of expressions as a form of natural language. A possible explanation for this is the lack of a simplified yet concrete domain into which students could map their expressions to understand their meaning in formal terms. This issue will be taken up in later chapters.

4.8 Learning Prolog by a Procedural Approach

The major difficulty in understanding the mechanistic view of Prolog is that it is very complex to grasp. Fine details can appear trivial or even tedious but are in fact crucial to a proper understanding of how the automatic mechanisms interrelate. It can often be difficult for beginners to know where to focus their attention whilst execution is taking place: an analogy is inspecting the inner workings of a clock to derive the time, taking
no notice of the clock face and the hands. In this respect, the use of a dynamic graphical tracer to illustrate selected events is essential for tutorial purposes (cf Rajan, 1985). At any given point during the running of a program, many activities are taking place all of which need to be tracked by the learner to derive a complete picture. For example, beginners may be distracted by the forward execution mechanism and disregard the possible instantiations which are also taking place.

Also, procedures in Prolog are collections of clauses where variables have scope only over one clause. Consequently, beginners often do not realise, or they confuse, the relationship of one clause to another within the execution process (e.g. only if a first clause fails will a second clause be tried). So, for example, in trying to comprehend recursive cases, learners forget that at each level in the recursion the first clause of a given predicate will be tried again, even though the recursive call may have been generated by (say) the third clause.

Another major problem with Prolog programs is that the distinction between procedures and data can be blurred due to the use of unification. But this means that it can be hard to work out exactly what the purpose of a given predicate is meant to be, particularly if the programmer has not selected appropriate predicate and variable names. Often one can only work out what is going on by mentally simulating what happens to a datastructure as execution proceeds. This is an extremely difficult task, and even Prolog experts may shy away from making definite assertions about what programs will do without the benefit of examining an execution trace.
Input Interpretation

Prolog code can be difficult to read from the procedural point of view for several reasons. Firstly, unlike other programming languages, there are no pointers to indicate flow of control through a program. In order to interpret a program a great deal of ancillary knowledge is required about Prolog's execution. For example, not only can backtracking take place, unbinding variables as it goes, but clauses can be used with different instantiations as input, producing sometimes radically different behaviour from that intended by the programmer. Secondly, for similar reasons, it is often unclear what in a Prolog program is a consumer of data and what is a producer. There are conventions - e.g. the last argument is often a variable which will be instantiated with the final output - but this is by no means always the case. Clauses can often be used with the last argument (the 'output' variable) instantiated, hence forcing the program to test whether a given datastructure would be a candidate solution. But even knowing that which arguments are to be bound to final output may not be informative with regard to the actual execution process: i.e. how the solution got computed at all.

Therefore, novices faced with Prolog code often find themselves at a loss to know how to begin to read it, and often feel that the code does not 'do' anything (cf Chapter 3).

Reasoning Strategies

The general level of discourse and the mechanistic discourse represent a view of Prolog as a more conventional programming language - i.e. expressions in the language are related to the
machine's actions. Because of this closer relationship to behavioural aspects of the learning situation, some aspects of the general discourse can be used to support understanding at the mechanistic level. For example, the sorts of temporal and causal reasoning which underpins understanding of events in the real-world can be utilised at the formal level: first do this, and then do that which will then cause some other event. Of course, the problem is that explanations of the 'cause' of events may be different in the two domains, which can often result in confusion.

For example, the 'member' relation in Prolog is defined as follows:

```prolog
member(X, [X|_]).
member(X, [_|Y]) :- member(X,Y).
```

which can given the following (rather simple) procedural reading:

'To find out if X is a member of a list first see if the head of the list matches X. If it does, then you have succeeded, and the rest of the list can be ignored. Otherwise, if this does not succeed, ignore the head, and recursively call member on the tail of the list to see if X can be found in the tail.'

The important point is that in this definition, no mention is made as to what to do when the empty list is encountered. Therefore, when the empty list is produced, the clauses simply fail. This behaviour is what is required, and the empty list is thereby dealt with by 'significant omission'. However, it is tempting to explain this failure with the rationalisation that there is no point looking for an object in an empty list. Whilst this is a perfectly sensible observation, it is not one which reflects the mechanistic discourse: it reflects either a logical view, or a real-world view. In other words, this 'explanation' tells us why the clauses have
been written the way they have, but it does not provide an explanation of what they do. This may seem a trivial distinction because if a student in class were to make the above observation, it would be accepted as indicative of an overall understanding of the situation. But if the student has not understood the mechanics of the execution, then this view could easily imply that Prolog 'knows' more about what it is doing than is really the case. This point will be further illustrated in empirical studies reported later.

Output

If students believe that Prolog 'knows' more than it does (cf Pea 1986), then we should certainly expect to see the kind of output observed by Van Someren (1987) noted above: i.e. instructions for humans to perform a task, rather than a computer. In this endeavour, students are liable to produce incomplete programs which have no robust underlying representation. Interpreting the machine's behaviour is also liable to be difficult if the student believes that Prolog has 'intentions' or, at the very least, its own method of 'doing things'. An inevitable consequence is a division in the student's mind between that behaviour evoked by her own program statements, and that due to the 'magical' automatic mechanisms. This can not only make debugging programs extremely difficult, but can also complicate prediction of behaviour.

Reflective Evaluation

Again, beginners are liable to use empirical evaluation techniques initially (i.e. by observing the computer's behaviour, the student establishes that the desired effect has been achieved). Whilst this
may serve for a while in the programming domain it eventually will not suffice to say that because a program has succeeded once (or even a few times) that it will continue to succeed in the future. Students should be working towards generality of solutions, and moving away from the context-dependent specific solution. In this case, there should be a shift, at some stage, from empirical toward hypothetical validation. A potential class of evaluation bugs, therefore, are those linked with a failure to predict possible outcomes, a situation which is particularly acute in relation to Prolog given the backtracking mechanism, and the possibility of obtaining multiple solutions. In this case, the student must distinguish between different categories of output: expected and correct, unexpected and correct, expected but incorrect, unexpected and incorrect results. This ability depends upon finely tuned discriminatory powers, particularly in the case where the computer turns out more results than expected: many students regard this as a general feature of Prolog programs, rather than a potential bug!

Students taught the procedural interpretation of Prolog are liable to become overwhelmed by masses of detail. Nevertheless, they do have access to a relatively well-defined concrete domain of computational mechanisms which they can use to structure their understanding. Their difficulties, therefore, are liable to focus around misapprehensions regarding the execution strategy.

4.9 Interactions between the two formal domains

It is frequently the case that Prolog novices pick up a hybrid interpretation, where both the declarative and the procedural views are used. We must, therefore, expect interactions between the
logical and the mechanistic domains. In this section we examine the
interactions between the two formal domains. It should be borne in
mind that the input for both the logical and the mechanistic levels
will be Prolog code - i.e. the same program can be read in both
discourses.

There is liable to be a two way interaction since it is feasible
that a beginner might misinterpret a logical expression as if it
had a mechanistic interpretation, or try to read a programming
expression as if it were logic - or both! Again, the novice is in a
tricky position because some assumptions will work, some will not,
and some are essential to make Prolog understandable at all. This
is because Prolog is not a proper logic programming language, and
therefore cannot be fully understood, or described, within the
logical domain of discourse.

Input Interpretation

In Chapter 3 the problem of interpreting declarative statements as
instructions to act was discussed. The possibilities here associated
with confusing logical and mechanistic interpretations where in
fact some mechanistic pieces of code cannot be given a logical
reading (e.g. expressions using the 'cut' operator, or 'read' and
'write' instructions) and some pieces of logic cannot be accurately
represented mechanistically (e.g. in cases where a correct
declarative specification cannot be turned into effective working
code).

Reasoning Strategies

Flow can go from logic to Prolog: i.e. deductive reasoning is
appropriate to both - but it cannot always go from Prolog to logic (i.e. procedural solutions which depend on changing facts).

Output

Prolog can be read as logic only if it has been purged: i.e. certain forms of expression denote a procedure which is meaningless at the logical level. Certain uses of variables also cannot be expressed as logic.

Reflective Evaluation

The validity of logical specifications can be tested according to behavioural evaluation, though this is not strictly part of the logical approach. However, lacking other more sophisticated knowledge, beginners may well need to validate logic programs using the behavioural criterion more appropriate to the mechanistic level (c.f. the discussion in the previous chapter). However, what reads as a good logical specification may not work when interpreted mechanistically, since a mechanistic interpretation may need to take into account the order in which actions are performed. In other words, the mechanistic account uses both the causal and temporal elements which are part and parcel of empirical/behavioural validation methods.

4.10 Discussion

The framework helps emphasise that the novice's contribution to the learning situation are assumptions and strategies which are rooted in the uppermost layer of the domain framework: the real world. This perspective is the one from which the other two domains (the logical and the computational) will be viewed and, perhaps more
importantly, actively interpreted.

This view indicates, therefore, that we should not be at all surprised to find novices using general purpose interpretation strategies in programming - i.e. we should expect them to use natural language interpretation skills to understand expressions in logic or programming, and to use practical reasoning skills rather than formal ones, and so on. The use of such strategies should be considered a 'natural learning pathway' (di Sessa 1982) rather than bizarre behaviour. By adopting the 'genetic', rather than 'formal' approach, the emphasis is upon what the students actually do in the learning situation, rather than on what they should do. This approach not only acknowledges the complexity of the novice programmer's task, but also recognises that a great deal of the beginner's behaviour is often insightful, even though from the formal learning perspective, it may happen to be inappropriate.

It is also evident that this way of interpreting programming performance helps to clarify the weaknesses (and strengths) of other studies of programmers criticised in Chapter 2. Those studies concern themselves with an intra-discourse analysis of learning - that is to say the success or otherwise with which a learner relates the various components of discourse within a domain. This would be a lateral movement across the components of the computational discourse for example. In other words, the process by which a learner arrives at an interpretation of a domain of discourse is outside the analysis; only when the student has arrived there is performance evaluated. In this situation, existing knowledge or previously developed strategies are not addressed by the theory. These studies do yield vital information with regard to
subsequent development and refinement of cognitive skills to the level of expertise, but their relatively narrow focus of study may ultimately distract us from the question of what it is that novices really need to do in the early phases of learning.

The above discussion has important ramifications both for language design and pedagogical practice. If it can be shown that the source for commonly occurring errors for beginners lies not at the level of the programming language and its operations, but are a result of assumptions and expectations derived from much higher levels in the reasoning process, then questions regarding 'learnability' and 'ease-of-use' of a language have a different focus.

The framework illustrates that, in principle, there are lots of possibilities for superbugs to arise. However, only two strands of difficulty in superbug effects for Prolog are examined in detail in the rest of the thesis, and they are: that associated with interpreting language, and that associated with interpreting the behaviour of the machine.

These two types of difficulties may be linked with the teaching approach taken. Students learning the declarative approach to Prolog, with its lack of emphasis on the mechanisms of the language, are liable to view Prolog as a cut-down version of natural language. 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 poor programs for several reasons. Firstly, declarative statements tend to be regarded as only descriptive (c.f. Ehrlich and Soloway, op. cit.). This may mean that programs simply do not
compute because clauses are badly formulated. Secondly, students are not inclined to use analytic comprehension processes, and therefore may not be forced to acknowledge which parts of the problem statement are crucial in the solution and which are not. Furthermore, programs are liable to be underspecified because inference processes associated with assumptions about language understanding are left implicit in program statements, rather than being made explicit.

Students introduced to the procedural semantics are less likely to make this type of 'linguistic' error, but may begin to interpret the execution strategies of Prolog as more intelligent than they really are. Assumptions about what would be the 'sensible' thing to do may obscure the details and hidden intricacies of the execution process.

More succinctly, the hypothesis is that in the absence of specific domain knowledge, declarative-style introductions to Prolog are likely to encourage beginners to use natural language understanding as a reference point for establishing the meaning of programs or expressions. On the other hand, students who are being taught procedurally with an emphasis on the mechanisms by which programs are executed are less likely to use natural language as a reference point, but may well use intentional concepts to explain the machine's behaviour.

Errors may occur, then, not only because the beginner misinterprets some aspect of a particular discourse (e.g. how to perform deductive inference) but also in trying to move from one discourse to another (e.g. transforming a piece of logic into a piece of
4.11 Conclusions

The framework helps illustrate the fact that the two apparently different views of Prolog (i.e. the declarative and the procedural) can be regarded as only a shift of emphasis within formal discourse. Prolog viewed as logic is a formal domain where program statements are interpreted as having a meaning according to the intended interpretation of the programmer. The constraints on this domain are broadly speaking, those of logic. Prolog viewed as a programming language is similarly a formal language, but the meaning of programs ultimately is mechanistic, and can be understood in terms of the behaviour they evoke from the computer. If students are to interpret Prolog programs properly, they should have an understanding of either, or both, of these levels. The contention in the thesis is that they frequently do not, but that they use knowledge and strategies from the real world to construct their own view of Prolog. This is a necessary, and often powerful learning strategy, but it can also lead to misapprehensions which are difficult to resolve.

Prolog's behavioural component is complex, and, because its syntax is non-committal, learners are tempted to hallucinate onto it whatever they think appropriate, rather than referring to an interpretation based upon underlying domain knowledge. The situation is complicated, because errors stemming from misconceptions at any of the levels will eventually show up in the behavioural component of the mechanistic domain - i.e. the computer does not do what the novice expected. In other words, some
misconceptions which are apparent at that level are likely to be due to interference from other levels. These are the bugs which have been defined in this chapter as superbugs.

Humans are sophisticated learning systems, and it does not take long for them to realise at one level, as they interact with computers, that there is not a 'hidden mind', or native 'intelligence'. But the strategies appropriate to human discourse are frequently the most appropriate to high level programming, and, quite naturally, particularly when under stress, these are unconsciously introduced into the programming domain. High level programming languages are designed to support high level problem solving skills. In doing this, expressions in programming formalisms are trying to fulful a dual purpose - to support an interpretation on the part of the human programmer which she can construe as relevant statements about the problem domain; but which at the same time, support the kind of mechanistic interpretation which makes the machine 'behave' in particular ways. This behaviour is in turn interpreted by the programmer as appropriate or otherwise to the task in hand. It is easy for inexperienced problem solvers to confuse elements of this complex situation, and programming in Prolog is complicated by the fact that the programmer herself must move between the two levels of interpretation, rather than being able to rely on the machine to play its role appropriately.

An important point to bear in mind is that because beginners have no specific domain knowledge, they are obliged to interpret a new situation from the general level of discourse. In this enterprise learners need to use general purpose discourse understanding and
reasoning strategies. Some of these strategies may be powerful and economical when they work, but they may equally introduce misconceptions and misinterpretations (cf Lewis and Mack, 1982). As they apply these strategies, learners will bring to bear any existing information they have which seems to them to be relevant. This information will be woven into a 'theory' or a set of explanations which constitutes their growing mental model of the programming language. The situation is complex, therefore, because learners cannot be expected to introduce only correct information (as presented in text-books and teaching materials) into their conceptualisations of the programming language. So, what may turn out to be a misconception at some point in the learning process may well have started out as part of an essential and powerful general purpose strategy.
CHAPTER 5

Methodology

While one might argue that cognitive psychology can tell us about problem-solving in a simple domain, one would be hard put to make a similar argument for complex problem-solving domains, such as computer programming. Given that there is precious little detailed, cognitive theory that would help us predict what problems novice programmers will have when learning to program, the first question that cries out for an answer is: 'What problems do novices have?' (p.1)

Soloway and Sleeman (1986)

5.1 Objectives

Recall that the objective of this thesis is to gather primary data to inform an analysis of the kinds of problems novice Prolog programmers have, and in so doing provide a partial answer to the question posed above. There are very few studies of what real Prolog novices do, and our understanding of their difficulties and problems is limited.

The study described here, therefore, involves observation of the early learning experiences of Prolog novices, to identify what happens when beginners are presented with the language, how they interpret it, and how they try to use it.

Novices, as has been argued, bring with them strategies for interpreting new situations and new information. A major point of interest is the extent to which these strategies work in the Prolog domain, or the extent to which they can lead to misperceptions. At this stage of analysis, the aim is to map out the size and shape of the possible space of errors that Prolog novices might have, to get a feel for the range of errors that they make. The discourse framework is used to help organise such observations. Later work would be needed to investigate in more detail the frequency and
persistence of observed errors.

5.2 Method

In pursuit of the rather general objective of studying novice programmers, several different data collection methods and techniques are needed. It has already been shown that there are no adequate theories of learning to program from which to generate hypotheses for controlled experimental investigations. It is also not clear at this stage in our understanding of novice programmers what advantage the strict experimental approach would have, since although experimental results may be robust, their generality is frequently limited, particularly in relation to novice performance (Sheil, 1981). As novices are involved in the process of learning, it is not possible to standardise groups, or tests, in ways which would make statistical evaluation of performance appropriate.

Video-taped think-aloud protocols are standardly used in Cognitive Science to investigate learning and performance issues (e.g. Anderson, 1982), and this was the basic data gathering technique used in the studies reported here. Since the research is exploratory, the methodology is structured clinical, a method also frequently used in this area (e.g. Perkins, Hancock, Hobbs, Martin and Simmons, 1986). In this situation, the experimenter sits with a student who is given a programming task. The major role of the experimenter is to observe, probing where necessary, and offering assistance if the student is having insurmountable difficulty. Subjects were encouraged to verbalise what they were thinking, but were put under no special pressures to do so.

Occasionally free protocols are used to make inferences about
learners' reasoning strategies, and at other times they are used to investigate expectations about performance which either derived from previous protocol analysis, or from findings in the literature. The two methods are complementary. Because of the atheoretical nature of this kind of protocol analysis, the data is interpretative, which has several drawbacks.

Firstly, it is very easy to introduce bias into interpretation, which then affects what kinds of issues are attended to by the researcher. At this stage in the collection of primary data, this seems inevitable. Eradicating bias would involve large research resources which were not available. The multi-levelled framework was developed during the data collection to provide some coherency for observations. Conclusions are therefore not definitive, but rather lay the ground for further work and greater refinement of data gathering techniques.

Secondly, it is inadvisable to generalise too far from protocol studies. Because of the intensive nature of protocol analysis only a small sample of subjects can be viably used, and, in the studies reported here, these are drawn from groups of students being taught particular views of Prolog, on courses run within the University. In this respect, they are not necessarily representative of Prolog learners elsewhere. However, again, analysis of these students' problems is a first step toward a more general perspective of Prolog learners.

Thirdly, the justification for free verbalisation in protocols reflects some of the concerns expressed by Schoenfeld (1983) about the way in which experimental situations affect the behaviour of
subjects. He claims that behaviour induced during video-taped sessions can be 'almost pathological'. Schoenfeld investigates how belief systems, the interactions with social or experimental environments, and control levels in decision making, can shape people's behaviour. Thus, perceptions or beliefs on the part of the subject as to the kind of activity they are meant to exhibit during experimental sessions affects the manner in which they work, and can prevent them from accessing relevant, known information from memory which would, in other situations, be easily accessible.

Acknowledging these difficulties, attempts were made to address them by providing a variety of settings in which protocols were taken. All participants were students at Sussex University, and the observer was known to them as a post-graduate student who ran 'surgeries' to help those having particular difficulties with Prolog. The observer declared herself at an early point during term-time as available for help, and confessed that she was not a mathematician, nor a computer scientist. Students volunteered to collaborate on the project, and were encouraged to do so by the course tutor. They were not paid for their collaboration.

A major anxiety for students was that there would be some 'hidden component' to the experiments: for example, hidden cameras, or especially devious tests which would weed out the people who 'really knew' how to use Prolog from the rest. The protocols were videotaped in a small room, with the camera clearly visible. The observer attempted to reassure students that what they saw was all that was happening, and that the 'tests' were not designed to identify weak students. The television monitor was turned away from view, so subjects could not see themselves on screen. Nevertheless,
students seemed most at ease when the camera was obviously only focused upon the terminal screen, and it could be seen to be so by the monitor. Thus, the later protocols consist only of the verbal discourse between observer and subject, and a record of the activities on the machine. Although some expressional data was lost, it was considered worthwhile since (a) it is not immediately obvious what contribution a record of students' bodily gestures would contribute to the analysis at this stage, and (b) students were clearly more relaxed. A writing pad was placed just to the right of the terminal screen so that written notes made by subjects were within view of the camera.

In order to try to reduce anxiety for one particular task (described below), some students were matched into pairs of roughly equivalent ability. They were presented with the task and asked to decide between them either what should be done or what it meant, with the proviso that they should reach agreement at the end of the session. This approach was stimulated by the discussion of 'constructive interaction' (Miyake, 1982) who argues that think aloud protocols of lone individuals do not yield the most interesting data. The explanations that one member of a couple offers to the other are likely to be much more informative about the various conceptualisations that individuals have. Furthermore, if there is disagreement, then interpretations can be challenged, providing a naturally occurring situation for more explicit specification of an individual's conceptual organisation.

However, social factors then are an added dimension to the protocols, which, in some cases, can contaminate the data. There may be cultural influences involved in this. The polite English
students who participated in these studies were very reluctant to accuse each other of being downright wrong. Instead, elaborate social deferences rendered some students entirely silent, whilst the other chattered on, or they went round in circles because they could not admit that each thought the other completely misguided. More consideration, perhaps, should have been given to friendship patterns within the group. We include only one of these studies in Chapter 8 because, in fact, although the social dynamics are very evident, it does provide a good illustration of abductive reasoning on the part of one student.

Finally, it should be borne in mind that individual protocols only provide 'snapshots' of students at certain points in the learning process. What appears to be an insurmountable difficulty for a subject during a protocol may turn out to be only a transient phase which is quickly overcome at a later date. The protocols, therefore, give a fragmentary view of the novice, which can sometimes over-emphasise spurious features of performance.

To help overcome this problem, longitudinal studies of two students were undertaken, providing the opportunity of observing the first 12 hours of learning Prolog. The methodology for these studies was adapted from Anderson, Farrell and Sauers (1982). They obtained videotaped protocol data on the first 30 hours of learning LISP:

The subjects worked with an experimenter who tried to do as little teaching as possible and let the student learn from the text. The main responsibility of the experimenter was to query the subjects about what they were thinking, why they tried various solutions, etc. However, if the subject had a serious misunderstanding or was lost in a problem, the experimenter would intervene with tutorial assistance.

We feel that we have a good record of the learning that was occurring in these sessions. Subjects were instructed not to
think about LISP when they were not in the experimental session. They were also not permitted to keep the textbook between sessions.

This method was followed, the textbook being replaced by the on-line teaching files used as Sussex University. The TEACH files in POPLOG are intended to be sufficiently complete for new students to learn programming languages in their own time, although when used as part of a course, additional support is given by tutorials, lectures and individual help.

Unfortunately, however, the students who participated in these studies frequently called upon the experimenter to answer questions, and provide explanation, which made it difficult for her to remain unobtrusive, and she was therefore sometimes unable to adhere to her role as 'independent' observer.

Expecting students not to 'think' about the language when away from the experimental situation was also rather naive. It may be that the student does not consciously sit down to work on the language at home, but nevertheless, one cannot help but ponder on the events occurring in sessions. It would also seem that a great many processes associated with skill acquisition can go on unconsciously, giving rise to insights away from the experimental situation. For example, on several occasions, students arrived at the beginning of a session with a realisation about what had been giving them trouble in a previous session. This would seem an inevitable process - students cannot be expected to confine their insightful behaviour to the laboratory.
5.3 Subjects

Group 1: Computers and Thought Students

The first group 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 centre mainly around the interference of natural language in interpreting Prolog clauses. (NB: For a brief description of course materials please see Appendix 2).

Group 2: MSc students

The second group of subjects were post-graduate MSc students, where 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 (POP-11), 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 mean, 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.

**Group 3: Case Studies**

The last form of data gathering was extended case studies. Two students participated: Alex and Lisa. Lisa was an MSc student and was therefore taking the Prolog course. She had no previous background in computing prior to coming to the University, but had learnt POP-11 in the previous term. Lisa attended 8 sessions of an hour each, bringing specific tasks from the coursework with her. In the event, the protocols from her sessions were not very informative mainly because, as she was taking the very intensive MSc course, her coursework took priority over other experimental tasks. It was difficult to get a sense of continuity out of the tasks she brought along, which were different from week to week, and as her progress was fast, the exercises varied considerably in
difficulty as the term went on. For this reason her case study is not reported but since she also participated in the MSc tasks, she is included as a subject there.

Alex was a post-graduate D.Phil student who had learnt other computing languages as part of his undergraduate degree. Alex attended 11 sessions, constituting the first 12 hours of learning.

5.4 The Tasks

Computers and Thought Students

It had been noted during teaching that several of these students seemed to think that so long as a program 'made sense' in English, then Prolog would interpret it correctly. They exhibited an apparent inability to understand the difference between what 'makes sense' to humans as a piece of text, and the formal description of a specification expressed in English. Typically, students found it hard to recognise the difference between what they had (incorrectly) written and the (correct) formal statement.

At first sight, there appeared to be two related components to the bug. Firstly, as Pea (1986) has pointed out, naive users often do attribute intelligence to the machine, particularly if it appears to be using natural language - they think the machine 'understands'. However, some students persisted doggedly in their belief that somehow the meaning of the words used in programs were significant to the machine. During one of the courses the tutor, being aware of this difficulty, made use of a fanciful 'martian' terminology in tutorials and lectures, insisting that every word in databases had to contain the letter 'm' somewhere - e.g.:
teamches(mfred, alisonm).
teamches(msusan, peterm).

Despite these considerable efforts students still acted as if they thought the computer would understand what was meant in a program, even though it was not expressed properly. Such students could describe their non-working programs quite adequately in English, but had no idea why Prolog made no sense of them. The solution to the programmed problem lay in understanding the English words, not in its structure. Thus, when shown programs which used complete nonsense words instead of predicates, these students were very confused.

By using natural language interpretation, then, it appeared that some students were falling into the trap of simply 'translating' their perception of a problem into a surface/syntactic version of the formalism with no recognition of the changed status of objects and relations in the formal domain. This is likely to contribute to a situation where fundamentally the user thinks that the formalism is much more subtle than it really is. The consequence in the programming domain is that the beginner is distracted from a problem-solving process into a translation process. The question of what function a piece of information plays in achieving a solution tends to be ignored. This can have serious implications for the kind of representation students select as appropriate for the task. It was also thought that this kind of problem would be more acute for those learners being introduced via the declarative semantics because of its lack of emphasis on mechanisms and processes.

In terms of the discourse framework, these students were following the same sorts of pathways as those subjects in deductive reasoning
tasks, as shown in Figure 5.1.

<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>REASONING</td>
</tr>
<tr>
<td>INTERP.</td>
<td>STRATEGIES</td>
</tr>
<tr>
<td>REASONING</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>REFLECTIVE</td>
<td>EVALUATION</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GENERAL</th>
<th>Implicit</th>
<th>Ordinary</th>
<th>Implicit</th>
<th>Empirical</th>
</tr>
</thead>
<tbody>
<tr>
<td>--&gt;</td>
<td>*---------</td>
<td>----------</td>
<td>----------</td>
<td>-----------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FORMAL</th>
<th>Explicit</th>
<th>Analytic</th>
<th>Explicit</th>
<th>Hypothetical</th>
</tr>
</thead>
<tbody>
<tr>
<td>--&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5.1 Pathway for Computers and Thought Students

Recall that in this case, students behave as if general discourse processes are appropriate at the formal level. In this case, it makes little difference whether the formal level is logical or mechanistic. The basic problem here is one of misinterpreting the nature of formal domains.

Our informal hypothesis was that interpreting Prolog clauses and rendering them into English would be relatively easy, since students would be moving from the unfamiliar formal discourse into the familiar general discourse. They would, therefore, be able to use their high level reasoning strategies to provide a 'sensible' rendering. On the other hand, the representation of English statements in Prolog might be more difficult, partly because there are many important aspects of natural language which cannot be represented directly in Prolog code, and partly because the students who participated in the study would be sophisticated in their use of natural language, and much weaker in their proficiency with Prolog. In other words, they would be moving from the familiar to the unfamiliar discourse in this type of task. The major focus
of interest was upon this latter process where the issue was the extent to which natural language could be seen to affect representation in Prolog.

3-Part Representation Exercise

An open-ended three-part test was designed with the simple intention of obtaining information about which part of the programming process appeared most susceptible to natural language interference – the interpretation of Prolog clauses into English, the representation of English in Prolog, or the debugging of Prolog clauses.

The clauses were presented on cards which the student turned over, one by one in their own time, writing their 'solutions' down. Nine students participated, and the protocols were taken during the fifth week of the course.

Part 1: Prolog to English

Figure 5.2 shows Part 1 of the task where Prolog clauses were to be rendered in English.

1. pencil.
2. female(sue).
3. cat(tom).
4. loves(mary, sue).
5. loves(john, henry).
6. mother(mary, sue).
7. father(john, henry).
8. parents(mary, john, sue).
9. parents(mary, john, henry).

Fig. 5.2 Computers and Thought Representation Task
Part 1: Prolog to English

This is a simple straightforward task since the clauses are not difficult and the mapping from Prolog to English is one-to-many -
i.e. the clauses can be correctly described in a number of different ways in English. However, we did expect students to point out that two place predicates could be read either way round depending on the programmer's intended interpretation (i.e. in Clause 6: 'mary is the mother of sue' or 'sue is the mother of mary').

Part 2: English to Prolog

The second part (Figure 5.3) consisted of English statements which had to be couched in Prolog. This is more difficult since the mapping is more restricted - only a limited number of Prolog clauses could represent the English sentences. Furthermore, these students were very new to Prolog, and so had little in the way of choice of representation.

1. There is a book
2. Here is a pencil
3. Jaffa is a student
4. Misha is a professor
5. Jaffa is male
6. Misha is a woman
7. Misha is clever
8. Misha teaches Jaffa
9. Jaffa admires Misha
10. Jaffa learns well
11. Jaffa works hard
12. Jaffa learns well because he works hard
13. Jaffa is a good student because he admires Misha so he works hard
14. All Misha's students learn well
15. Any of Misha's students who work hard will succeed

Fig. 5.3 Computers and Thought Representation Task
Part 2: English to Prolog

Part 2 of the task initially presents a variety of simple statements (1-11). But the statements become increasingly complex, and begin to include causal links: e.g. 'because', 'so' - which cannot be directly represented in Prolog. Subsequent clauses also
include English quantifiers (e.g. all and some) which Prolog also deals with indirectly. This task pushed students rather hard into the issue of how a formal language could possibly represent statements which in English are relatively simple, but which contain concepts which are outside of Prolog's limited range. Observation of their reactions to this difficult task would provide a small window into their perception of Prolog, and the extent to which they thought it could, or should, be able to function as a discursive language, and how subtle they imagined its scope to be.

Clauses 12 - 15 created so many problems for students that it was decided to omit Clauses 4 and 5 to help reduce the demands of the task. Later protocols, therefore, do not contain responses to Clauses 4 and 5 in the test.

Part 3: Buggy Clauses

The last set (Figure 5.4) consisted of clauses having a mixture of syntax and variable bugs of various sorts which students had to identify:

1. knows(michelangelo, rafael)
2. hates(Michelangelo, Rafael).
3. admires(rafael leonardo).
4. Famous(leonardo).
5. friends(michelangelo, [leonardo, botticelli]).
6. friends(michelangelo, [leonardo, titian]).
7. parents(jim, couple(mary, fred)).
8. composer(Bach)
9. composers[back, hadyn, handel].
10. artist(X): painter(X).
11. heard_of(X, Y):- knows(X, Y),
    knows(Y, Z).

Fig. 5.4 Computers and Thought Representation Task
Part 3: Buggy Clauses

These are, again, straightforward.
These tests were unpiloted, being used as a device to initially engage students in the observational studies and to begin creating Prolog clauses. In the event, Parts 1 and 3 presented no significant problems for these students, and their comments as they worked were minimal. These two parts of the test are therefore not included in the main discussion of the findings. However, the protocols, and copies of subjects' written answers are provided in Appendix 3 for reference purposes.

The rather motley collection of English statements in Part 2 of the task were not perceived by students to be necessarily contributing to a 'proper' program. Had more resources been available, a more carefully structured set of statements would have been devised to probe translation issues more selectively, providing a clearer insight into the interplay between natural language and Prolog.

The experimental set-up for this kind of study similarly needs very careful consideration. It may be, for example, that some students viewed the session very much as an 'experiment' and, under these conditions, they perceived their task to be creating clauses in Prolog in whatever way they could. They may have felt, therefore, that it was not 'legal' to say that it could not be done. Subjects also had to write down answers on paper - had they been asked to write them in a real program which they could run, some errors would naturally have been corrected, or might never have arisen.

A further difficulty is the fact that prior to doing the English to Prolog set, they did the Prolog to English set. It is possible that there was carry over into the second exercise from the first. The observations reported in Chapter 6 suggest that it would be
appropriate to devise a programme of experimental situations to present a more robust, empirical validation of the effects of natural language interpretation on program construction, and these experiments should include a much larger sample of subjects.

The analysis of observations in this study is reported in Chapter 6.

Interpreting Clauses and Abduction

As mentioned earlier, six of the Computers and Thought students were matched into pairs of roughly equal ability on the basis of their success with the previous task. These couples were presented in the laboratory with the standard clauses which define 'membership', but with the meaningful predicate name taken out:

```
foobaz(X, [X|_]).
foobaz(X, [_|Y]) :- foobaz(X,Y).
```

Each pair worked together and were instructed that they should discuss the clauses, and reach agreement regarding their function so far as was possible.

They had pursued a Prolog course for seven weeks prior to attempting this exercise, and had been recently introduced to the notion of recursive list processing. List processing clauses are typically rather hard for beginners to express in English. The original aim of the exercise was to ascertain how much information they could derive from the syntax of these clauses with the knowledge they had so far.

In the event, however, as noted above, social factors rendered most of these protocols useless. However, we have included one by Julie and Mary because it illustrates a case of abductive reasoning. This
protocol is discussed in Chapter 8 and the protocol transcript can be found in Appendix 4.

In terms of the framework the protocol provides information about the move represented in Figure 5.5, where an intuitive leap is made from the mechanistic input to general level reasoning strategies.

<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse --&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>REASONING</td>
</tr>
<tr>
<td>GENERAL</td>
<td>Implicit</td>
</tr>
<tr>
<td>MECHANISTIC</td>
<td>Explicit</td>
</tr>
</tbody>
</table>

Fig. 5.5 Pathway for Abduction in Interpreting

Prolog Clauses

It will be seen in the discussion that the student carefully considers this move, and justifies it in a coherent way. Furthermore, in the process of defending her interpretation, she makes reference to other concepts which are appropriate to the mechanistic level, drawing on experience with a previous exercise. The reason that it suggested she has moved into the general level of discourse is not because of the specific content of what she says, but the way in which she uses that information to weave a plausible story.

The protocol is reported in Chapter 8, and the discussion there supports the notion that although abductions sometimes lead to error they are not random or snatched from thin air, but are instead often sensible lines of reasoning which have their basis in
experiences where things have worked in the past.

**MSc Students**

A major part of learning to program in Prolog is appreciating how to control the inference, unification (or matching) and backtracking processes during execution. Execution is therefore complex, and predicting events by mental simulation can be extremely difficult. A great deal of time was spent with MSc students in free protocol sessions asking them to describe the execution of different types of clauses (e.g. recursive list processing, matching structures etc.). This produced large quantities of protocol material, which, because of its relatively unstructured nature is not directly reported here. However, the experience of working in close collaboration with these students informs the more speculative discussion sections to be found in the data analysis. Protocol data which is reported was obtained from an experiment associated with understanding backtracking.

**Backtracking Exercise**

Coombs and Stell (1985) had identified a problem for students associated with one of the automatic mechanisms - backtracking. The major point of their analysis was that if students are uncertain about the procedural semantics, then debugging is made difficult because errors are attributed to program specification, whereas the error may simply lie in the order in which clauses have been written. Backtracking is particularly difficult:

> Many of the major problems of Prolog debugging are related to the process of backtracking...Even a simple program of two or three clauses may backtrack in complex ways which, if represented in full, would occupy many pages of trace. Such
behaviour may be difficult to predict without a detailed mental symbolic execution of the program text. However, mental execution is difficult to perform accurately given the lack of syntactic markers to serve as signposts and the need to relate information widely distributed in the sequence of execution events. (p.3)

Using the program in Figure 5.6, they identified two particular

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: ?- a(X).

Fig. 5.6 Coombs and Stell's (1985) Backtracking Example

bugs which they call 'redo body from left' (hereafter RBFL) and 'try once and pass' (hereafter TOAP). RBFL is characterised as thinking that backtracking proceeds from left to right instead of right to left. Thus, upon failure of the instantiation of c(1) in Clause 1, students think that Prolog goes immediately to the sub-goal d(1) in Clause 2, unbinds the variable, and instantiates it to 2. In fact Prolog will return to the subgoal e(1) to attempt to re-satisfy, before backtracking to d(1).

The second error - TOAP - stems from the assumption that having satisfied sub-goals d(1) and e(1), which lead to subsequent failure in Clause 1 because no c(1) can be found in the program, Prolog immediately passes on to the second clause for satisfying b(X) in Clause 3 without any attempt at re-satisfying the sub-goals in Clause 2. The effect of this bug, then, is that backtracking does not take place at all.

As a test instrument, the program is not ideal. Firstly, because
there is only one possible instantiation for 'c', students can, by meta-level reasoning, identify what the eventual outcome is without bothering to go through the steps in the execution. The first subject in the experiment in fact avoided the task altogether through use of meta-level reasoning. In subsequent protocols, therefore, the experimenter made it very explicit to subjects that it was the course of execution which was required, not the overall result.

Secondly, the use of a single variable throughout the program can allow students to gloss over the issue of instantiations between clauses. Since Prolog clauses are only locally scoped, each clause could have used a different variable name, but this would not affect the execution, since unbound variables would share. Presumably the authors intended to simplify the task by using only one variable name, but it is not clear whether or not students are more confused by this uniformity than they would be by obvious distinctions between variables in each clause.

Initially volunteer MSc students were asked to participate in protocol studies to run this experiment to see if any of them demonstrated TOAP and RBFL. Individuals talked through the program whilst being videotaped. The program was written on a large card to make pointing easier. The experimenter, who was familiar with the program, provided help when it was clear that subjects were simply getting lost in their description. Thirteen subjects participated in the study, five weeks into their Prolog course. After each protocol session, students were individually debriefed, and their errors explained to them in some detail by the experimenter.
Because of the relationship of the cut operator to backtracking, the experiment was repeated with an amended program to include the cut, and an additional clause to make meta-level reasoning a little more difficult (Figure 5.7). The same procedure was used, and eight of the original group of students

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).

Fig. 5.7 Modified Coombs and Stell program participated a fortnight after the previous test. It was expected that learning would have taken place after debriefing of the first experiment which proved to be the case. It would have been preferable to have run the manipulation on a new group of subjects, but unfortunately none were available.

However, the protocols obtained from both these experiments were not only informative about backtracking misconceptions, but provided an extremely rich source of information about many other kinds of errors. For this reason, the discussion of findings in Chapter 7 is separated into two parts.

The first discusses the particular errors identified by Coombs and Stell, providing an analysis of TOAP and RRFL, and the effects on interpretation of the cut. This discussion is confined to the mechanistic discourse and the view of the task is represented on the framework in Figure 5.8.
Students are given a program which they must interpret to predict its behaviour. Although we know that students must use the general level in this process of understanding, no special attention is paid in this part of the discussion to the influence of the general level of discourse. The main focus of attention is upon the students' ability to correctly relate different components of one discourse level.

The second part of the discussion then takes a higher level view of the task, and points out general features of subjects' performance as they begin the task of working through the program. This is represented in Figure 5.9.
<table>
<thead>
<tr>
<th>Name of</th>
<th>Components of discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discourse</td>
<td>INPUT</td>
</tr>
<tr>
<td></td>
<td>V</td>
</tr>
<tr>
<td>GENERAL</td>
<td>Implicit</td>
</tr>
<tr>
<td></td>
<td>--&gt;</td>
</tr>
<tr>
<td>MECHANISTIC</td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td>--&gt;</td>
</tr>
</tbody>
</table>

Fig. 5.9 Dual Discourse Analysis of Coombs and Stell task

where inter-discourse effects are considered. This analysis describes the use of high level reasoning strategies in the programming domain and identifies possible causes for errors such as TOAP and RBFL. Protocol transcripts for both tests can be found in Appendix 5.

The Case Study

During the case study Alex covered the basic core of Prolog: facts and rules; database accessing; pattern matching; backtracking; recursion; list processing; and use of the cut. Example programs to illustrate these features were a family tree program, a database of artists and their works, a parts inventory for bicycles (Clocksin and Mellish, 1981, p.54) and the development of this program into a simple parser. Execution of programs was illustrated through the use of a dynamic tracer which is part of the FOPLOG system, and the Byrd box models (Byrd, 1980) which give an overview of control flow through clauses. Alex also did the first of the backtracking exercises given to the MSc group.

All of these topics were taught via the TEACH files, except for those exercises drawn from Clocksin and Mellish (1981).
Although the methodology was based upon that of Anderson et al. (op. cit.) there was a problem. Alex had a difficult time learning Prolog because of the interference of his prior knowledge of other languages. He seemed to be resistant to many of the fundamental principles of Prolog, and certainly was not a passive recipient of 'correct' information. He was, rather, attempting to understand Prolog concepts in terms of what he already knew, and became frustrated when he could not make it fit. His attempts at understanding were further hampered by the fact that the experimenter was not familiar with the other languages he knew, and was unable to confirm whether or not an analogy was appropriate.

This situation can be regarded as the computer expert's version of superbugs. That is to say, whereas novices have superstrategies (potentially giving rise to superbugs) which stem from the general discourse level, experts may have alternative computer related discourses which interfere in much the same way with the acquisition of a new language - in particular a new language which seems to be fundamentally different from ones previously known. Similar processes of trying to make sense, abducting explanations and defending conclusions drawn during learning (whether correct or not) are observable.

This case study, then, provides an interesting account of the sorts of problems that experienced programmers have. The discussion in Chapter 9 highlights the long term effects that this form of superbug has on program construction. Complete protocol transcripts are not provided for this study due to sheer bulk, but transcripts of the programs he wrote are to be found in Appendix 6.
5.5 Conclusions

The table in Figure 5.10 provides a summary of the subjects with whom we worked, and our particular focus of interest for each group.

<table>
<thead>
<tr>
<th>Subject Pool</th>
<th>Focus of Interest</th>
<th>No. of Subjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computers &amp; Thought</td>
<td>Interference of natural language in clause generation</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Interpretation of Clauses and Abduction</td>
<td>2</td>
</tr>
<tr>
<td>MSc</td>
<td>Predicting machine behaviour: Coombs &amp; Stell (TOAP/RBFL)</td>
<td>13</td>
</tr>
<tr>
<td>MSc</td>
<td>Predicting machine behaviour: Coombs &amp; Stell with Cut</td>
<td>8</td>
</tr>
<tr>
<td>Case Study</td>
<td>Long term effects of prior knowledge on learning</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 5.10 Summary of Observational Studies

Although there are methodological difficulties in studies of this type, they do provide a basis for further development and analysis. Using these methods in combination with the multi-levelled framework, it is possible to provide a coherent account of the bugs and superbugs found in the performance of the novice programmers observed. However, it is only a partial account, and there are inevitable gaps in the analysis which need filling by later work.

5.6 A Note on Presentation of Data

In sections which present programs or program fragments, clauses are numbered Clause 1, Clause 2 etc., as follows:

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).

......
These numbers are not part of the program, but are for reference purposes only. In the discussion, protocol fragments are presented, and lines spoken by subjects are numbered on the left, e.g.:

17 Uh huh. Yes, right....ok.....ahhh...
18 Yes well, uhm...so it goes to rule number 2
19 and er and it finds b(X) if d(X) and e(X)...

This again is for ease of reference. The line breaks are not significant, being on the whole determined by a computer program which breaks text into an approximate line length, and inserts the numbers on the left. However, where this process made reading difficult, minor textual adjustments were made. Transcripts of protocols are to be found in the appendices, although illustrative fragments are included as appropriate in the text.
CHAPTER 6

Natural Language Superstrategy:

Observational Studies of Computers and Thought Students

6.1 Introduction

This chapter reports upon the observational studies of nine Computers and Thought students and addresses the issue of natural language superstrategies in the programming domain. The discussion focuses on the ways in which interpretation processes appropriate to natural language understanding interfere with representation and interpretation at the formal level of the discourse framework. The types of difficulties which the natural language superstrategy can lead to bear close relationship to Pea's (1986) egocentrism bug where the student believes her program is as meaningful to the computer as it is to herself, and the sorts of difficulties reported in the deductive reasoning literature.

Computers and Thought students are not introduced to detailed descriptions of Prolog's automatic mechanisms, nor are they are taught formal logic. However, they do need to be able to represent clauses in a structured, consistent way to allow Prolog to correctly interpret programs. Having only a limited amount of domain-specific knowledge, therefore, it is likely that the influence of ordinary comprehension processes, and the use of English to provide 'meaning' will be apparent.

The points of emphasis in this chapter, therefore, are that:

1: Novices are likely to use natural language interpretation in the interpretation of Prolog clauses because they have little choice.
2: This superstrategy can be harmless - or even of great value to
the novice - if concepts from natural language are used as a
'knowledge frame' (di Sessa, 1985 op.cit.) - e.g. 'predicates are
verbs, arguments are nouns'.

3: The superbug which can arise, though, is when students leave
implicit assumptions in program statements - i.e. when an analytic
interpretation has not been considered, and the computer is assumed
to be able to perform the necessary inferences to interpret the
statements as intended by the programmer. This in turn affects the
kind of structure and representation used in programs.

The protocols of Computers and Thought students performing Part 2
of the representation task, described in Chapter 5, will be used to
illustrate various facets of the interference of English in clause
creation.

Recall that these students were being introduced to Prolog by the
declarative approach, and the test was given to them in the fifth
week of their Prolog course. A major concern in programming,
whether in Prolog or not, is that data or information should to be
structured 'sensibly' for the current purpose, whatever that may
be, or else programs will produce meaningless or incorrect answers.
Learning how to sensibly represent knowledge is arguably the most
important issue to be confronted by the beginner in learning to
program. Programming also involves the notion of problem solving,
which often means devising an algorithm which the computer will use
to work out the answer to some problem.

The programming behaviour of the Computers and Thought groups
seemed to resemble that described by Pea (1986) - i.e. they were
showing signs of thinking that Prolog could 'understand' programs
which were incomplete - and it was also thought that students were
not making appropriate decisions about representation issues in
their programs. Since they were not provided with any strong logic
background, and their knowledge of the mechanistic semantics was also rudimentary, it seemed inevitable that they would use natural language interpretation to some extent. The simple presence of natural language interpretation was not necessarily, therefore, problematic, but the extent to which it might eventually contribute to serious difficulties was of concern.

It will be shown below that students did indeed use the superstrategy of natural language interpretation in the task of creating Prolog clauses. This strategy proved, on the whole, to be very useful to most of the students, but also gave rise to, or contributed to, other misapprehensions which will be elaborated in this chapter. Figure 6.1 is a list of the superbugs identified from the studies.

* Natural Language Superstrategy
* Translation Superbug
* Causal Reasoning Superbug
* 'Prolog is Intelligent' Superbug

Fig. 6.1 Natural Language Superstrategy and Superbugs

The Translation and Causal Reasoning Superbugs stem from an implicit assumption that Prolog code can be read as a restricted form of English and that it has a correspondingly similar range of expressive power. The 'Prolog is Intelligent' superbug is premised on the tacit notion that the computer is more subtle and sophisticated in its interpretative capabilities than is really the case.

The main effect of these assumptions was to distract students from the activity of problem solving, and creating structured representations, into re-representing the English in a kind of 'Prolog-ese'. If students think that programming consists of a
surface translation from English into the hieroglyphs of Prolog, they can avoid the crucial issue of knowledge representation which is relevant to all aspects of computing and AI. Such students will confuse themselves by writing disorganised programs which will either take a long time to run, or which will not run at all. Students may then have to waste time debugging badly conceived programs. Furthermore, re-representing English text in Prolog is not necessarily going to solve the given problem if no attention has been paid to issues regarding algorithms.

Because Prolog syntax is rather non-committal, however, it can be relatively easy for the optimistic novice to hallucinate onto it a reading which often seems to reflect more about natural language semantics than it does about Prolog's execution strategies. It can be hard for beginners to devise robust explicit representations because they muddle English (in this case) and Prolog. This is a complex problem which revolves around the need for analytic comprehension not only in the reading and writing of programs, but also in interpreting descriptions of Prolog clauses provided by teachers.

Since this is a preliminary study of language use in novices the test was open-ended. Therefore, some parts of the discussion are supported by protocol fragments, whilst others are more speculative.

6.2 Observations of the 3-Part Representation Exercise

The Prolog to English clauses (Part 1) presented very few problems, except where students were unfamiliar with the form of Prolog
assertions such as: 'pencil.' Students proceeded more or less silently through this phase of the task, writing down their interpretations. The only comments were to indicate that clauses such as 'loves(mary, sue)' could be interpreted either as 'Sue loves Mary', or 'Mary loves Sue'. Students also had no trouble with the buggy clauses (Part 3), rapidly spotting all the errors. All of the 9 students who participated in the study provided correct or adequate responses to all of the questions (see Appendix 2).

Van Someren's (1987) observation noted in Chapter 4 may be pertinent here as a partial explanation of this successful performance: in Parts 1 and 3 of the task subjects were presented with clauses couched in Prolog, which may have helped limit and constrain the task and the range of interpretations which could be made by subjects. Figure 6.2 illustrates the framework pathway to represent these parts of the task.

<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse———&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>REASONING</td>
</tr>
<tr>
<td>INTERP.</td>
<td>STRATEGIES</td>
</tr>
<tr>
<td>GENERAL</td>
<td>Implicit</td>
</tr>
</tbody>
</table>
be rendered into the terms of the familiar general level (English). This move did not seem to present many difficulties. This may be due to the rather simple range of clauses which were used. For example, list processing or recursive clauses can be very difficult to express in English. Unfortunately, this group had not covered this ground after only five weeks of the course.

Part 3 of the task, which involved spotting errors in clauses, also presented little challenge, but again, the range of possibilities was limited. The errors were mainly syntactic and the effective performance observed from all subjects who participated in the study may simply illustrate the effective use of rote learning by keen students!

Since these parts proved to be almost trivially easy no protocol data is presented but students' responses can be seen in Appendix 2.

Part 2 of the exercise - English to Prolog - on the other hand, proved much more interesting, although the success rate (in terms of providing a reasonable representation) was still high. The protocols for this part of the task provided a good deal of evidence to suggest that English was being used in the programming domain in ways which could lead to misconception. The pathway in the framework for this part of the task is shown in Figure 6.3.
Figure 6.3 shows a basic idealised pathway for this task, where the input is natural language (INPUT), which is reasoned about at the general level (REASONING), but then a move is made to the mechanistic level (REASONING) to generate explicit expressions in Prolog (OUTPUT) which are evaluated at that level (EVALUATION). There may be extra mappings up from the mechanistic level to the general level in the output and evaluation columns but these have been omitted for clarity in this diagram.

The pathway observed, however, is shown in Figure 6.4.
the mechanistic level and worked instead almost exclusively at the general level of discourse. Nevertheless, as will be seen, many of their responses are reasonable and adequate (i.e. they would run as program fragments). Part 2 of the exercise is reproduced in Figure 6.5 below.

1. There is a book
2. Here is a pencil
3. Jaffa is a student
4. Misha is a professor
5. Jaffa is male
6. Misha is a woman
7. Misha is clever
8. Misha teaches Jaffa
9. Jaffa admires Misha
10. Jaffa learns well
11. Jaffa works hard
12. Jaffa learns well because he works hard
13. Jaffa is a good student because he admires Misha so he works hard
14. All Misha's students learn well
15. Any of Misha's students who work hard will succeed

Fig. 6.5 Computers and Thought Representation Task
Part 2: English to Prolog

It can be seen that Clauses 1 - 11 have a very simple structure, and, to some extent, it would be difficult for students not to represent these in what might be described as a reasonable form (remember that these students were novices so some amount of generous interpretation was appropriate). Clauses 12 - 15 build upon these simple expressions, but involve the use of causal links (i.e. 'because' 'so') and naive quantification (i.e. 'all' and 'any'). So the fact that students turned out representations which were reasonably uniform and which could feasibly run as parts of programs is not really the main focus of attention. The main issue is how they described what they did, and the sorts of justification offered for their actions as they constructed the clauses. It is this analysis which indicates that they were following the pathway
shown in Figure 6.4 rather than that shown in Figure 6.3. Superstrategies often lead to success, but equally can lead to superbugs. In order to refine the concept of superbug, it is essential to identify points at which superstrategies degenerate, providing fertile ground for misconception. One such point is where a student provides a correct response, but for the wrong reasons.

It has been argued in earlier chapters that if a programming language is used as if it were a natural language then the result is incompleteness in programs because background assumptions are left implicit in the code. Examples of how this can come about are provided below.

**Translation Superbug**

Part of the natural language superstrategy seems to be an assumption that English grammar can be used as a model for understanding Prolog clauses. The superbug which can arise - here called the Translation Superbug - is to begin regarding Prolog as a form of natural language where the range of expression is thought to be similar to English, and Prolog is expected to have corresponding constructs to English.

The point at which this kind of consideration became apparent was from Clause 10 onwards - i.e. sentences of the form 'Jaffa learns well'. This was the first clause which was neither a simple description of an individual (e.g. Misha is clever), nor a relationship between two objects (e.g. Misha teaches Jaffa). Three students commented directly on the slightly odd status of 'well' and 'works hard' in Clauses 10 and 11 respectively.
Subject 1

11 Jaffa learns well
12 well we haven't come across adjectives at all
13 I could do Jaffa learns, but not Jaffa learns
14 ... Well I suppose you could just put learns
15 well bracket jaffa
16 that's how I'd do it ...
......

WRITES: (Clause 10) Learns well (Jaffa).

28 ... I'm not too sure about these terms ...
[referring back]
29 of works hard I've not come across anything
30 like this... its an adverb isn't it, well?
31 yeah, and we haven't used that
32 kind of structure before

WRITES: (Clause 11) Works hard(Jaffa).

Subject 3

21 Well it's like I don't remember coming
22 across adjectives before ... if that's what 'well' is
23 It's an adverb ... so I was thinking
24 I could do learns well jaffa
25 to sort of keep it all into the one
26 word, but I was just wondering
27 if there was something I'd forgotten ...

WRITES: (Clause 10) learns-well(jaffa).

Subject 5

3 Well it's gone into an adjective ...
4 you know and usually its just two nouns
5 and a verb ...

WRITES: (Clause 11) works_hard(jaffa).

Clearly, these students were using natural language grammar to interpret the task, and, as pointed out earlier, this is not necessarily problematic. Despite a few syntactic problems in the construction of predicate names (e.g. Subject 1 included a space between words, and Subject 3 used a hyphen where presumably she intended an underscore character) the representation is fine. But the following protocol illustrates how Subject 3 slid into difficulties from this sort of interpretation.
Example of Translation Superbug: Subject 3

The comments which accompanied Subject 3's answers to Clauses 1 and 2 were interesting, since they were the first signs that she was using English grammar as a reference point for Prolog clauses:

Clause 1:
6 There is a book... phew...
WRITES: there(book).

Clause 2:
14 Here is a pencil ... actually we haven't done any with 'there's and here's' and things.
15 And we haven't used words other than like verbs,
16 I don't think ... likes ... loves ... that's it
WRITES: here(pencil).

Clause 10 produced a long pause and the comment:

21 Well it's like I don't remember coming
22 across adjectives before ... if that's what 'well' is
23 It's an adverb ... so I was thinking
24 I could do learns well jaffa
25 to sort of keep it all into the one
26 word, but I was just wondering
27 if there was something I'd forgotten ...
28 another way of doing it.
29 We've started 'Sir' in the lectures but
   [a teaching exercise]
30 we haven't done anything in the
31 program where it's language ...
32 I don't know if you could do jaffa learns well on
33 the inside of the brackets with a comma
WRITES: learns_well(jaffa).

Clause 11 is similarly dealt with: works_hard(jaffa). This student decided to change the 'because' in Clause 12 to an 'if' (more fully discussed below). She hesitated though for a moment about which way round to express it:

46 jaffa learns well ... yuk ... because ...
47 well ... I don't think I've seen a 'because'.
48 I could change that to
if he works hard he learns well ...
which would be the nearest I've come across I think ... um ... if he works hard he learns well if he works hard ... I'll have to do it in that order ...
learns well jaffa if works hard jaffa.
Which is the meaning of it anyway.  

WRITES: learns_well(jaffa) :- works_hard(jaffa).

This hesitancy is further illustrated as she attempted Clause 13, and ended with a comment on the 'validity' of the predicate 'good_student' which is related back to 'what we do' in English:

"jaffa's a good student because he admires misha so he works hard ... haven't got good_student ...
E: Is that a problem?"

It's just different I suppose ...
what would I do ... I'd do because he admires ... well I can't do because so I'd do if ... so what the clause would be is he's a good student if admires misha and works hard ...
no hard and admires misha ...
no the other way round is the best meaning of it ...
admires misha ... if he admires misha ...
good_student if admires misha and works hard or works hard if admires misha ...
works hard if admires misha
good_student ... mmmmm, I'll go back to the first one. But good student seems daft.
I mean we never put an adjective in front of something ...  

WRITES: good_student(jaffa) :- admires(jaffa, misha), works_hard(jaffa).

It is interesting that this student chose to focus her anxiety on the predicate 'good_student', a trifling matter in comparison to the hidden complexities of the expression associated with causal links. But her formulation for Clause 14 began to show how disregard for Prolog's interpretation of clauses can lead to difficulties. She commented:

"All misha's students learn well ...
don't tell me they all admire her ..."
76 all misha's students learn well ...
77 learn well student misha

WRITES: learns_well(student, misha).

This clause was trivially incorrect because the term 'student' should be a variable (i.e. beginning with a capital letter), but it also appeared not to mean what she thought it did. The clause as written means something like: the 'learns_well' relation holds between the objects 'student' and 'misha', whereas she had intended it to mean Misha's students learn well. It is as if the comma between the two objects in the body of the clause (acting as a separator) was being read by her as the word 'of'. The same situation occurred in the following clause:

Clause 15:

78 Any of misha's students ...
   [pause]
79 succeed student misha if
   [pause]
80 it won't work can't put ...
   [mutter]
81 ... umm ...
82 if work hard student ... hmm that'll do.

WRITES: succeed(student, misha) :- work_hard(student).

This was meant to represent the statement: Any of Misha's students who work hard will succeed. An adequate representation for this clause might have been:

\[
\text{will_succeed}(X) :- \text{teaches}(misha, X), \text{works_hard}(X).
\]

where 'will_succeed(X)' would cause Prolog to search for a clause matching 'teaches(misha, X)' as in Clause 8: 'teaches(misha, jaffa)' - and then to search for the 'works_hard' clause with whatever instantiation had been obtained by the success of the previous subgoal (e.g. if X had been instantiated to 'jaffa' as in Clause 8, then Clause 11 could match with 'works_hard(jaffa)'.
Clauses 14 and 15 as written by the subject conflate these two separate operations.

The interpretation of Subject 3's error, then, is that the distinction between constructs in natural language and constructs in Prolog became blurred. This led her to construct a Prolog clause which she interpreted sensibly, but only because she read the comma as 'of'. Since the comma appears as an innocuous syntax item with no obviously defined function, it is open to misinterpretation in this way by novices. In this case, the student has performed a surface translation of the English statement into Prolog-like form.

Recall the pathway used to illustrate this type of error (see Figure 6.6).

<table>
<thead>
<tr>
<th>Name of Component of discourse</th>
<th>Discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>REASONING</td>
</tr>
<tr>
<td>INTERP.</td>
<td>STRATEGIES</td>
</tr>
<tr>
<td></td>
<td>EVALUATION</td>
</tr>
<tr>
<td>Implicit</td>
<td>Ordinary</td>
</tr>
<tr>
<td>General</td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td>Analytic</td>
</tr>
<tr>
<td>MECHANISTIC</td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td>Hypothetical/</td>
</tr>
<tr>
<td></td>
<td>Empirical</td>
</tr>
<tr>
<td>MECHANISTIC</td>
<td>Implicit</td>
</tr>
<tr>
<td></td>
<td>Empirical</td>
</tr>
</tbody>
</table>

Fig. 6.6 Subject 3's Pathway for Clause Construction

It can be seen that the appropriate mappings down into the mechanistic level to ensure that her clauses would run as programs are not made by Subject 3. Her justifications and reasons for producing the particular Prolog clauses that she did referred either to English grammar, or what makes sense in English. Reasoning at the mechanistic level might have led her to consider how Prolog would interpret clauses in terms of its underlying
mechanisms (e.g. how the matching process would work, how forward execution would progress, and so on). Nevertheless, as is implied by the pathway shown in Figure 6.6, despite remaining at the general level, what she did in her protocol had coherency, but the end result was not appropriate as a program.

So, the difficulty with Subject 3's strategy is that concern to represent English meaning obscured considerations about underlying structure for clauses. As students progress to more complex examples of Prolog clauses, they may find that during this early phase of learning, they have not been laying the foundation for an understanding of the kinds of representations which are typically used in declarative programs.

Of course it may simply be that, in this particular protocol session, Subject 3 did not explicitly reveal knowledge which she had of the mechanistic domain, but which nevertheless informed her actions. This issue is taken up below in the discussion of the related Causal Reasoning Superbug.

Causal Reasoning Superbug

The 'causal reasoning superbug' involves the expectation that Prolog can be used to represent causal events just as English can. It is an example of assuming that Prolog has similar constructs to English and a similar range of expression, but causal links are of particular interest. It was pointed out in Chapter 3, in relation to formal reasoning in logic, that causal relations do seem to provide an important basis for the derivation of meaning in natural language, but that neither logic nor Prolog can directly represent causal or temporal events. Clauses 12 and 13 include a 'because'
and the open-ended question was: what would students do?

A domain appropriate response to Clauses 12 and 13 from either logicians or expert Prolog programmers might be that they simply cannot be properly represented because the respective formalisms do not address causal events. However, the enthusiastic and co-operative students in our study all did something, with varying degrees of anxiety apparent. Subject 5, quite reasonably, wondered whether Prolog had a 'because' operator, just as it had an 'if' and an 'and' operator:

Subject 5

9 Is there a special symbol for 'because'? ....

This subject, having been told there was no 'because' operator, then came up with the notion of replacing the 'because' by the ':=' operator (reading it as 'if'). The clause was then interpreted as: if it can be established that Jaffa works hard, then it can be assumed true that Jaffa is a good student. Clause 11 stated that Jaffa did in fact work hard, so the program captured the implicit fact that Jaffa was a good student.

22 I want to put and he works
23 hard, but because ... if I just put
24 because it's going to be another ... it's
25 not going to relate to the two

E: OK. We don't have a Prolog sign for because so we can't use that.

26 But there's one for 'if', is that right?

E: Yes

27 If he works hard then he learns well.

.....

WRITES: (Clause 12) learns_well(jaffa):-works_hard(jaffa).
This solution involves some knowledge from the mechanistic domain - the notion of 'causality' is derived from Prolog's execution strategy.

All students eventually adopted this basic technique, some writing it down immediately without comment (cf Subject 2's protocol) others pondering for a while. Whilst evaluating whether or not to substitute an 'if' for 'because' Subject 4 and Subject 6 only referred to the different English meaning of the two:

Subject 4

21 mmm ... you don't think
22 of because and if having the same
23 connotations

E: Right ... would you be able to say why?

24 Mmm.
 - pause -
 25 There's something more cert ... 
 26 there's more certainty with
 27 because ... you know with an 'if'
    [hand gesture]
 28 you know, possibilities, whereas
 29 a 'because' is more certain

WRITES: (Clause 13) good_student(jaffa) :- admires(misha),
                                      works_hard(jaffa).

Note that Subject 4 omitted the second argument in the first of his subgoals - i.e. he forgot to put who admires misha.

Subject 6

WRITES: (Clause 12) learns_well(jaffa):- works_hard(jaffa).

23 I'm not sure if this is quite right,
24 but er ... that's the only way I can
25 think of putting it ...

E: That's fine. What are you worried about?

26 Mainly because it's um ... um ...
27 because he admires misha so he er works hard,
28 which isn't quite the same as and he works hard ...
29 he works hard because he's
admiring misha ... not and he works hard, so
... I don't think that makes
quite the same English sense ...

Furthermore, having decided to use an 'if' several students became confused about which part of the sentence should come first, i.e. in which direction the dependency relationship lay (cf also Subject 3's protocol above):

Subject 9

1 Mmm. These are difficult

E: Right.
What's hard about it?

2 I know 'and', and 'if'
3 but ...
4 Um ...
5 Could you use if?

[nods]
WRITES: (Clause 12) works_hard(jaffa):-learns_well(jaffa).

E: Ok, so read out to me what you've written in Prolog.

6 Er, Jaffa works hard if jaffa learns well

E: Right, and ...
[laughter]
what do you think the English means?

7 Um. The English of this,
8 what I've written?

E: No, this English, sorry
[pointing]

9 Jaffa learns well because he works hard ... um
10 Yeah ...
11 Should be the other way round ...

E: Mmm
Do you think?

12 Um ... Jaffa learns well if jaffa works ...
13 Yeah ... it might make more sense

This uncertainty is presumably related to the use of 'if' and 'then' in English. The clause can actually be left as it is with
the substitution of ':-' read as 'if' in place of the 'because'.

Jaffa learns well :- he works hard

The implication goes from right to left. However, an if/then statement, as more usually expressed, has the condition on the left side and would read:

IF Jaffa works hard, THEN he learns well

This difficulty seems to correspond to an error noted in the logic literature where people commonly misinterpret the direction of an implication. In Prolog this manifests itself in misreading a clause such as:

    a(X):- b(X).

as 'if a(X), then b(X)'.

This in turn has effects which appear to give rise to a form of causal reasoning. From the above misreading, students are in effect saying: 'Will this happen?' rather than 'Does this logically follow?' more clearly illustrated in the sentence: If I tip my cup then my tea will fall out. This kind of causal reasoning requires an intuitive temporal dimension (one thing must happen before another). Prolog viewed as logic has no such dimension, but Prolog viewed as an executable programming language does - the way Prolog actually works through a program allows for a non-linear (because of the backtracking), but plausible sense of time. If students are consistent in this somewhat unorthodox interpretation of the ':-' operator, they will probably progress quite unaware of their misconception for some time, though of course, their understanding of example programs, or programs in text-books, will be limited.

It was interesting to note that whilst most students fretted about the 'because/if' substitution, they quite happily omitted the
quantifier 'all' and and the word 'any' in Clauses 14 and 15. There may be several explanations for this. For example, students may have been secure in their knowledge from previous experience of Prolog that variables can be used to cope adequately with naive quantification, and had no qualms about the omission. The confrontation with 'because', however, would be novel to them. Alternatively, it may be that using variables in this way less obviously alters the meaning of the English sentence than does the 'because/if' substitution.

It is clear that the students were using effective reasoning strategies to determine what to do in the circumstances. Since they were not participating in courses teaching formal approaches to logic or programming, we would not expect them to be able to make the relatively sophisticated statement that some expressions cannot be represented adequately in Prolog when asked to do so in a protocol session.

Part 2 of the Representation Task confronted students with the relationship between English and Prolog in a rather unsubtle manner. In some respects, the mapping between reasoning about the task at the general level, and reasoning at the mechanistic level was being constructed during the exercise by those students fretting about the substitution of 'if' for 'because'. This mapping is represented on the framework in Figure 6.7 (the vertical line between general reasoning and mechanistic reasoning) but the wavy line from there to general output indicates that the expression generated contained implicit information
Fig. 6.7 Pathway for Causal Reasoning in the Representation Task

and evaluation of the expression ultimately took place at the general, not the mechanistic, level.

Nevertheless, as was pointed out earlier, it was not 'success' or 'failure' which was of most interest in these studies, but rather the methods by which students arrived at an answer at all. The causal reasoning component of the task illustrated how general reasoning strategies can be used to incorporate information from both the general and the mechanistic levels to produce a viable — if not formally correct — solution. The real superbug here would be to systematically assume that Prolog somehow had knowledge of the causal relationship which is believed to hold between, say, two sub-goals. This view would be particularly damaging for a student, for example, who was trying to read the declarative program of the River Problem shown in Chapter 3. Understanding how the program worked would be almost impossible since there are no opportunities to establish causal links.

'Prolog is Intelligent' Superbug

Thinking that Prolog is intelligent is an extremely easy assumption
to make, and can confuse novices in terms of the degree of explicitness required in programs. This issue was discussed in Chapter 4, and an example is provided from the protocols. The following comment was made by a student examining the clause:

parents(mary, john, henry).

Subject 7

5 Actually I never quite got to grips
6 with these three things in brackets

E: Mmmm ... in what way?

7 Well I never quite understood why, um,
8 you can know it's Mary and John who
9 are the parents of Henry and yet you
10 haven't said specifically ... you know I
11 don't understand that the computer knows
12 that that's what you meant ...
13 Rather than that Mary is a parent
14 of John and Henry ... cos it doesn't
15 Y'know think that necessarily ...

In this case, the student was confused by the fact that Prolog apparently 'understands' what the programmer has not explicitly stated - which variable represents what kind of object - he had not yet come to grips with the process of pattern matching. Yet, at the same time, in other situations the programmer is required to represent what to the novice are 'obvious' things, e.g. in the definition of 'brotherhood' that an individual should not be his own brother. The point that may be hard to grasp is that Prolog does not 'understand' anything about parents - the programmer hallucinates a 'meaning' onto clauses. As far as the computer is concerned, of course, correctness of interpretation is linked with structure, not content.

Other factors contributing to thinking Prolog is intelligent may be the vocabulary associated with resolution, non-determinism, and
'selection' of clauses. The concept of non-determinism may be unfamiliar to some beginners, and the language used in elementary text-books to describe those aspects of Prolog's behaviour which relate to resolution can be very confusing. A non-deterministic mechanism can be misinterpreted at this elementary level as one which can be relied upon to 'choose' the correct candidate for resolution from amongst a set of possible candidates.

Text-books and teachers often describe points at which Prolog adopts the depth-first branch in tree traversal as a 'choice-point'. Prolog is described as 'deciding' to take the left-most branch (Clocksin and Mellish 1st edition p. 71) or it 'picks' the correct rule (Clocksin and Mellish, 2nd edition, p 83). Beginners can be forgiven for attributing 'intelligence' to Prolog if they do not understand either the notion of non-determinism, or the fact that Prolog is not really 'choosing' at all - if it were intelligent enough to choose, it would surely be intelligent enough to choose decent possibilities!

This particular superbug was also evident in the studies of MSc students, and the discussion will be taken up again in the next chapter.

6.3 Discussion

The tasks described above have demonstrated a tendency for beginners to use natural language as a model for determining the creation of and interpretation of Prolog clauses. Whilst this in itself is not necessarily detrimental, examples have been provided to illustrate how students can become sidetracked from the activity of structuring data into the activity of translation. In effect,
students may become more concerned with the linguistic model of what clauses 'mean' than with the pragmatic concern of what Prolog will interpret them as in the context of a running program. Understanding the meaning of English sentences in a problem statement is only the first step towards creating a program.

The major problem seems to be that if students interpret expressions in the formal domain of programming using natural language skills, because of the lack of analytic interpretation, programs are liable to be incomplete. Also, English renditions of clauses may be loose, and therefore misleading. Prolog clauses can be rendered in English in several different ways, but only some of those ways will accurately reflect what the Prolog clauses actually represent. Take the following definition of brotherhood for example:

\[
\text{brother}(X, Y) :\neg \text{male}(X), \\
\quad \text{parents}(\text{Ma}, \text{Pa}, X), \\
\quad \text{parents}(\text{Ma}, \text{Pa}, Y).
\]

with the interpretation 'X is the brother of Y if X is male, and X and Y have the same parents'. It is very easy to slip into incorrect descriptions such as 'X and Y are brothers'. This becomes particularly evident when queries with unbound variables are used:

\[
?- \text{brother}(X, Y).
\]

Using standard Prolog descriptive language, this would be interpreted as: find me an X and a Y between whom the brother relation holds. However, a beginner might be tempted to interpret this as:

find two brothers

or

X and Y are brothers
or

find me the brother of X

which of course are all incorrect, since the test of 'maleness' applies only to X. The object represented by the variable Y could be female, in which case, X is her brother. This problem is a combination of (a) unpacking what we actually mean in English by some notion (here of 'brotherhood') and making it explicit (which occasionally causes some surprises), and (b) interpreting the Prolog syntax in strict accordance with this explicit representation of the relation as operationally defined in the program. English does not often have to cope with quantities of unknown objects (i.e. unbound variables) and novices seem frequently to get into difficulties with them. Further examples of this sort of problem involving variables is provided in the discussion of the case study in Chapter 9.

But there is another potential area of difficulty in relation to natural language which has not been explored by protocol study, but is the subject of speculation in the discussion below. The use of natural language can disguise the presence of the underlying mechanisms in a rather subtle way. Teachers and pupils have to communicate about Prolog in natural language. Both, therefore, have to make the kinds of implicit assumptions which are required to understand the teaching discourse. In this enterprise, however, it can be very easy for beginners to infer what Prolog does from the natural language description offered, and not by analytic examination of the clauses being referred to. So, whilst the teacher is talking, the learner is construing meaning from the verbal discussion, and is co-operatively hallucinating onto some
Prolog clause the particular interpretation which it is the intention of the teacher to communicate. But when the novice is left to her own devices to recreate this interpretation, she has only the rather uniform Prolog syntax to work from. It is hard to recreate all the natural language descriptions which were used to make it interpretable by the teacher, particularly when the particular words were common English words, which would not normally stand out as being of particular significance. For example, the standard first clause to define the 'append' relation looks as follows:

\[
\text{append([], L, L).}
\]

Assuming that novices at different stages during learning may only have a hazy grasp of Prolog's automatic mechanisms, there is very little here to cue them as to the function of this clause. The only clue is provided by the predicate name, which was devised by the programmer. What, then might be regarded as significant? Apart from the predicate name, the novice might note that the empty list stands out as perceptually different from the other two arguments. If she is observant, she might also notice that since the last two arguments are the same, some sort of matching process is implied. But otherwise, the clause is fairly inscrutable.

An English interpretation of this clause which 'makes sense' of it might be:

\[
\text{append the empty list TO a list TO PRODUCE a list}
\]

In this case, it is (again) the apparently insignificant comma which has, as it were, been expanded to provide not only a 'sensible' reading of the clause, but which also now contains implicit directionality. The significance of the pattern match can
now be deduced: appending the empty list to another list produces an unaltered list. Being able to provide this reading, however, is dependent on other knowledge - that very often the first two arguments in Prolog clauses are input variables, and the last is an output variable.

The point is that the teacher may automatically provide verbal interpretations such as this without recognising to what extent they are dependent on a really quite full understanding of the execution mechanisms. The student, on the other hand, can understand the above English interpretation, and feels that she has therefore understood the Prolog representation, perhaps without analysing in sufficient detail how the clause achieves its effect. The student may know what the clause does - because its function is captured in the meaning of the English words used to describe it, and the 'meaning' of the predicate name - but she has no idea how it does it. As was pointed out in earlier discussion, since clauses eventually have to be understood in terms of the behaviour they evoke from the machine, the novice is now stuck because she feels she understands, but then is unable to compose the syntax so as to produce the effect she requires because she lacks knowledge about the 'how' component of the task. The chances of her being able to correctly construct clauses which run as intended at this stage in her learning are minimal.

This may be a plateau on which many Prolog novices get stranded for quite a long time, because neither the teacher, nor the student can quite pin down where the problem lies. This view of the role that natural language plays in program interpretation may also go some way toward explaining another common phenomenon - that students
find themselves unable to make any headway into the interpretation of a program, but when a Prolog expert, or the teacher, is called upon to provide a reading, the program suddenly seems almost trivially easy. This can be a very frustrating experience. Debugging programs will also be difficult if there is a disparity between what students think their program is saying (described at a linguistic level), and what it is actually doing (at the mechanistic level).

6.4 Conclusions

Some simple examples of the interference of English in the process of reading and writing Prolog clauses have been considered. The summary table of responses to the last 5 clauses of Part 2 of the exercise (Figure 6.8) illustrates that although students may have reasoned about issues related to the general level of discourse during clause construction, their eventual offerings (generously interpreted) are basically good.
<table>
<thead>
<tr>
<th>Clause</th>
<th>Subject Number ----&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>V</td>
</tr>
<tr>
<td>10</td>
<td>P</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>P,V</td>
</tr>
<tr>
<td>13</td>
<td>P,V</td>
</tr>
<tr>
<td>14</td>
<td>P,V</td>
</tr>
<tr>
<td>15</td>
<td>P,V</td>
</tr>
</tbody>
</table>

**Fig. 6.8 Summary Table of Answers to Last 5 Questions**

**Key**

* = basically correct representation  
P = syntax error in predicate name  
V = syntax error in variable  
I = incomplete clause  
SB = superbug

It must be borne in mind, though, that an important aspect of the observation studies was not simply to look at the results of reasoning but to examine by what means students arrive at conclusions about representation. From the comments in protocols, it is clear that many of the students in the study used the natural language superstrategy, in the event, mostly to good effect.

These observations confirm, at this initial stage of analysis, that Prolog novices in the early stages of learning are inclined to behave as if Prolog is 'similar' to natural language. This view is consonant with observations of untutored people working in the domain of logic: the different underpinnings for formal and natural language are not perceived, leading to non-analytic interpretation. Causal links are considered to be very important by the novice, and attempts are made to represent them in Prolog, where the process of
execution provides an intuitive temporal dimension.

But the most important ramification is that students were able to get along using their natural language skills, and it is only under circumstances such as those in the protocol sessions above that it becomes obvious that there are potential underlying misapprehensions. Many students are likely to pass on from this view of Prolog fairly rapidly, but for some students, minor misapprehensions can turn into major difficulties. When teaching Prolog, therefore, it should not be assumed that students who appear to be coping are always coping for the right reasons. Prolog's flexibility may in fact, for some learners, turn out to be a trap, and teachers of Prolog must be aware that the use of natural language as a model of Prolog has its dangers.

This chapter has focused on the role of natural language in the programming domain, and the effect this has on representation. However, this group of students were being presented with a particular view of Prolog, a view which in some senses led them into certain potential superbug situations. The next chapter focuses on a group who received quite a different introduction to Prolog, where the emphasis was upon the execution domain, and interpreting Prolog's behaviour.
CHAPTER 7

Meta-level Reasoning Superstrategy: Studies of MSc Students

7.1 Introduction

This chapter describes the two studies of MSc students describing the execution of a program and focuses on their understanding of Prolog's automatic mechanisms, in particular backtracking, and their ability to predict behaviour from program statements.

Originally these studies were stimulated by the report of backtracking misconceptions by Coombs and Stell (1985), and the intention was to 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, as was pointed out in Chapter 5, 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 two types of analysis. The first section of the chapter is a discussion of the findings based on a single discourse analysis: i.e. from within the mechanistic discourse, the specific bugs identified by Coombs and Stell are discussed, and a collection of other bugs which were observed are presented. The term 'bug' is appropriate here because the discussion deals with mismappings between components within one discourse level. Bugs identified are shown in Figure 7.1.
Bugs in Mechanistic Discourse:

* 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

Fig. 7.1 Bugs in Mechanistic Discourse for MSc Students

These bugs are discussed first in order to clear the ground for the second section which provides a dual discourse analysis. In that part of the discussion the influence of general discourse processes on interpretations of the program is considered. This type of analysis, it will be argued, provides a much more coherent picture of what students are doing, and the bugs identified in the single level analysis are then set in this broader context.

Recall that MSc students have had a term's previous experience with computers, and have learnt POP-11 already, so some interference from one language to the other was to be expected. They were introduced to Prolog at the mechanistic level, but were also doing logic as part of their knowledge representation course. In contrast with Computers and Thought students, they are less likely to have naive problems with natural language interference. The task they performed was also of quite a different nature, focusing on the computer's behaviour rather than representation issues. This is a difficult task in Prolog because of the complexity of the underlying mechanisms. During mental simulation and verbal report of execution, therefore, beginners may well have to rely on intuitive strategies to cope.

One such superstrategy is to consider the computer as a full
participant in discourse processes. Using this approach, the student reasons at a meta-level about what would be a 'sensible' thing to do in the circumstances - i.e. what a person would do. This superstrategy is similar to Pea's (1986) intentionality bug, but it will be shown below that the strategy is a reasonable one, used by most students participating in the studies. The superstrategy and its main superbugs are listed in Figure 7.2.

* Meta-level Reasoning Superstrategy
  * Identity Superbug
  * Wishful Thinking Superbug
  * Left-to-Right Bias Superbug

Fig. 7.2 Meta-level Reasoning Superstrategy and Superbugs

The basic assumption which seems to underpin these superbugs is that Prolog behaves with the same sorts of insight and common sense as a person. Pea's (1986) comment, that this point should not be taken in too facile a manner, is worth re-iterating: the assumption is often tacit and can be seen to affect students' behaviour even though they might deny thinking that Prolog has any foresight.

The major points in this chapter are:

1: Predicting machine behaviour from Prolog programs is difficult.

2: As well as the low-level misconceptions to be expected in this area, students are susceptible to superbugs when their superstrategies go awry. These superbugs are usually associated with making assumptions about Prolog behaviour which save them having to go down into minute details of the execution.

3: These strategies can lead to confusion partly because they prevent the learner from finding out what actually happens during execution, but also because they lead to the assumption that Prolog can do much more than it can.

4: Other high level assumptions contribute to misinterpretations of the machine's behaviour, and these seem to be related to students' reading habits.
A cautionary note is in order in this section. As pointed out in Chapter 5, students were simply asked to verbally describe the execution. But this task is actually very complex, because students were being asked to read and understand the program themselves, and then to articulate - in natural language - the very precise activities of the execution level. Just as students make implicit assumptions in the writing of their programs (which can lead to incompleteness), so they are likely to make the same kinds of assumptions in their verbal reports. In other words, upon occasion it is difficult to know whether students omitted to mention a step in the execution because they assumed it to be obvious, or whether the omission indicated that they really were unaware of that step.

Also, because of the comparative laxness of informal verbal report, sometimes students used a turn of phrase which implied a misconception, but which perhaps should not be taken too literally. In the discussion of the data only those cases which appear quite strongly to suggest an underlying misapprehension are reported.

7.2 Section 1: Mechanistic bugs

This section describes the findings from the backtracking exercise described by Coombs and Stell (1985) viewed from the mechanistic level, as illustrated in Figure 7.3.
<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse-----------------&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>INPUT</td>
</tr>
<tr>
<td></td>
<td>VINTERP.</td>
</tr>
<tr>
<td>GENERAL</td>
<td>Implicit</td>
</tr>
<tr>
<td></td>
<td>MECHANISTIC</td>
</tr>
<tr>
<td></td>
<td>--&gt;</td>
</tr>
</tbody>
</table>

Fig. 7.3 Single Discourse Analysis of Coombs and Stell task

That is to say, the INPUT is a program, the REASONING strategies are appropriate to the mechanistic level, the OUTPUT required is verbal, but should be as explicit as possible, and EVALUATION of that output will be hypothetical (i.e. because students were not allowed to run the program, their explanations of what would happen would have to be evaluated according to their theory of the Prolog interpreter and its activities).

The program to be interpreted is shown in Figure 7.4.

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: ?- a(X).

Fig. 7.4 Coombs and Stell'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' sub-goal first - i.e. backtracking is perceived to progress from left to right instead of right to left.

Broadly speaking the presence of TOAP and RBFL errors was confirmed in some of our students (see Figure 7.5). But these rarely occurred on their own, unconfounded by other misconceptions.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No. of TOAP:</strong></td>
<td>6</td>
</tr>
<tr>
<td><strong>No. of RBFL:</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td>4</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>13</td>
</tr>
</tbody>
</table>

Fig. 7.5 Results of backtracking task

We first discuss examples of TOAP and RBFL, and then examine the other errors which were identified.

**Try Once and Pass**

A clear-cut example of TOAP is shown in Subject 9's protocol. This student had participated in a case study some two months previously, and was 'recalled' to take this test. He had not, therefore, been using Prolog for several months. However, he began confidently:

Subject 9

1 You've given X as a goal
2 so it'll start at the top of the database
3 and it would say X if b(X) and c(X),
4 a(X) if b(X) and c(X)
5 and so it would look at the subgoal
6 and it would see b(X) if d(X) and e(X)...
7 and so it would look at the subgoal and
8 go away and look for d(X)
9 and it would find d(1)
10 so it would bind X to 1
11 and - does that bind X to 1 for the whole clause?
12 It does - so you get d(l), e(l)
13 and it goes through to see if it can find an e(l)
14 and it can,
15 so it succeeds on Clause 2 -
16 its found a b(X) - does that feel right?
17 b(l) if d(l)....yeah yeah ok
18 so it goes away and pops back up to here
   [indic. Clause 1]
19 and looks at the next subgoal
20 and it goes blot, blot -
   [indic. past Clause 2 and Clause 3]
21 are these all currently instantiated to 1?
   [indic. variables in Clause 1]
22 They are, aren't they,
23 so they're going forward to look for a c(l)
24 and, surprise surprise, it doesn't find one!
25 So its got to backtrack
26 and look for further instantiations of b
27 and it finds that b(X) if f(X)  <- TOAP
28 so it goes through and looks for an f
29 and it finds an f as 3,
30 so it succeeds with b f as 3...
31 and so that's that instantiated to 3,
32 a(3) if b(3) and c(3).

Subject 9 was one of the only cases of TOAP which was uncompounded
by any other errors. More typical was Subject 4 who seemed at first
as if he would avoid it:

Subject 4

35 and now it tries to satisfy c(X)
36 and it looks down looking for clauses for c
37 and we get c(3)
38 so it's going to fail
39 because X had to be instantiated to 1 there.
40 So we have to backtrack
41 and try and satisfy b(X) again.
42 So we've failed on that clause,
43 we'll have to try on this clause -
44 hang on do we backtrack there
45 or can we backtrack to ...
46 uhm I'm not sure whether we backtrack
47 to the last choice it took on one of the sub-goals
48 um. ....Now I think,
49 no because it's gone all the way through b(X)
50 I think we have to backtrack to
51 the last choice it took with b(X).
52 So um if we do that
53 we take the second clause  <------ TOAP
54 then we've got b(X) if f(X)
55 and trying f(X) as the subgoal now,
56 we get \( f(3) \).

Redo Body from the Left

A clear case of RBFL was found in Subject 5's protocol. At this point he had tried to succeed with \( c(1) \) and had failed:

Subject 5

43 So it'll fail on that.
44 So it'll then backtrack
45 and see if it can find anything else for \( b(X) \).
46 So back at the rule here we have to
47 get rid of the - wherever we were -
48 I don't know - I was on d, yeah, d -
49 I'm getting lost now....
50 Yeah \( d(1) \) so we try \( 2 \) now - \( d(2) \)  
51 and \( e \) is going to instantiate the same
52 and it can, it succeeds, so, fine,
53 so \( b(X) \) comes \( b(2) \)...

Having illustrated clear cases of TOAP and RBFL, we now discuss the other mechanistic errors observed in the protocols.

Multiple values for 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 \( e(1) \). Therefore, at this point, the goal to be proved is \( e(1) \). Typically students thought the goal was \( e(X) \).

Subject 11

15 so \( X \) gets instantiated to 1.
......
21 So it's at this stage it moves on
22 to see if it can instantiate -
23 if it can satisfy \( e(X) \)
24 and it finds that it's number 6 - that \( e \) -
......
29 so it would get \( e(X) \) instantiated to 1...
30 uhm but...with \( X \) being a variable there...
31 I don't think that matters.
32 I think the 1's ok there as
    [indic. \( e(X) \) in Clause 2]
At this point, the same thing happened with the 'c' goal, which should be instantiated to c(1). The student thought the goal was c(X), and thereby X in this goal was allowed to instantiate to 3:

36 it goes back and tries for c(X)
37 uhm it doesn't get that till rule -
38 er well fact 9
39 and its X becomes instantiated to 3...

So at line 39 X was thought to be representing both 3 and 1. The student was aware, however that he was confused:

41 Am I getting confused with these X's? ...
42 Because, ah right,
43 because it's the same variable isn't it?

It is sometimes difficult to tell whether these errors are slips of language - i.e. simply faults in the description - or whether they are indeed bugs. For example, when this subject had another go at it, the description he offered might either be an inarticulate - but correct - description of unification, or there may be residual signs of the bug indicated by the implied comparison between X in one clause and the instantiation of X in another:

45 It, uhm..if its all the same variable
46 right...look here, it takes....
47 Can I start again?
48 a(X), b(X), c(X), so it comes in a(X)
49 and it goes down and it matches here
50 which calls b(X)
51 which calls d(X) which takes 1,
52 uhm b(X) uhm which is 1...
53 so at this point it fails
54 when it finds that c(X) is 3
55 Uhm, ok.
56 Cos this, cos this was a 1
57 this was a 3.
58 So it would come back with 'no'.
Subject 8 also stated quite clearly in line 30 that X was instantiated to 3, and yet it seems in lines 36-37 that the success with 'c(X)' was regarded as merely co-incidental:

Subject 8

24 So another subgoal is set up, f(X).
25 That's right, so uhm,
26 then it scans this part of the program
   [indic. facts]
27 and ends up with a value of 3,
28 3 instantiated to X,
29 so f(X) succeeds and so rule 3 succeeds,
30 so er X is instantiated to 3
31 and so the subgoal succeeds up here
   [indic. Clause 2]
32 sorry....er ......

EXP: there...
   [indic. b(X) in Clause 1]

33 Yes er .. well although its gotta have...
34 er... yes, so that this subgoal succeeds
   [indic. b(X) in Clause 1]
35 but um....
36 Yes I suppose that c(X) would succeed too
37 because um the same value here 3

Of course in other circumstances, different X's in independent clauses could instantiate to different objects, so students are not entirely misguided in their interpretation. The use of X throughout, as mentioned in Chapter 5, may simply confuse subjects. The major point, though, is that if students are unclear about variable instantiations, then they will have difficulties understanding backtracking.

Another manifestation of the multiple values bug is when students think that the X's are straightforwardly bound to different values. An example of this is contained in the protocol for Subject 2, but because this problem arose in the context of another bug - the database bug - this protocol is described below.
The Database Bug

It was pointed out earlier that interference effects were expected between POP-11 and Prolog, and the most obvious example of this 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' 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 (but without the backtracking facility), the database itself is a global variable whose value is a list of lists. Matching procedures are written into programs which manipulate the data in the database. So the difference is not so much in the surface appearance of the two types of database, but in the underlying interpretation that the respective compilers make of objects in the database. A list in 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 (although see Chapter 9 for a case where it proved intractable). 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. 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).

Subject 2 showed signs of the database bug, which also had effects on what he thought the instantiations of variables were, which led him into the multiple values for variables error. He called Clause 2 'a procedure' which again is a word used in Prolog programs to describe a collection of clauses which define an operation. But it became clear later in the protocol that his sense of the term 'procedure' was more the POP-11 variety:

Subject 2

5 and then it would call this procedure b(X)
6 which would look then at the premisses of that argument
7 which are d(X) and e(X)
8 and would call the first
9 which is d(X)....um
10 and then it would instantiate X in d(X) to 1,
11 no, because 1 is the first one there
12 um and then it would look at e(X)
13 and it would instantiate X again to 1
14 because e is the first one there
15 and that's a 1.

There was some uncertainty about what X in the head of the clause
should have been, which he worked out, although he remained a little confused:

17 I'm not quite sure then what that
18 X would be instantiated to
19 because there's not,
20 let me see that would be a 1,
21 and that would be a 1 so um,
22 um so that that is saying b(X)
23 if d(X) and e(X)
24 and it finds those
25 instantiated to 1,
26 so I take it that X would be instantiated to 1.
27 I'm not terribly sure about that actually,

For a moment, he thought that having completed Clause 2, execution would pass on to Clause 3, but this was corrected:

28 but it's next call at any rate would be
29 would go back to there
30 and it would find another call of b(X)
31 which is here
32 and it would be able to call that
33 if there was f(X),
34 and it would find f was 3
35 um....oh actually it might not do that
36 because if it's already instantiated that in fact
37 it wouldn't go on to do that
38 ...so then it would call..
39 because its already been able to do b(X)
40 it would only call second b(X) if it
41 couldn't do that particular one,
42 so then it would go to call on c(X)
43 and there is no procedure c
44 so it would go straight to the database
45 and find c was 3
46 um and so it would instantiate that X to 3
47 um and it would succeed
48 but I'm not quite sure what that X
49 would be instantiated to
50 because you have a 1 there and a 3 there,
51 and there's no um...
52 there's nothing to say they should be
53 added together or anything....

Lines 52 and 53 implied an assumption that some procedure would act upon the values of variables to produce the final output - as is frequently the case in POP-11 programs. But he realised that there was a problem with regard to final output:
54 So I'm not quite sure what that X
55 would be instantiated to,
56 presumably to the first one which would be a 1,
57 but um I'm not sure about that.

The tendency to view Prolog programs as if there was a significant
difference between facts and rules was quite common and may be the
result of teaching strategies. A typical illustration of this bug
at work is contained in the following extract where the subject
checked whether or not the solution to the query '?- a(X). ' was
simply present in the database. Even if it were, Prolog would not
be able to jump straight to that 'fact' without having worked down
the program. Since the head of Clause 1 matches the input query,
this rule would be tried first even if there was a fact involving
'a' lower down:

Subject 12

1 Ok it's trying to prove a(X)
2 to do that it's got to prove
   the subgoals b(X) and c(X).
3 So it looks to the database -
4 rather, first of all it looks to the
database to see
5 if a(X) is there and it's not.
6 There's no functor in the database.
7 So it tries to prove b(X),
8 it looks to the database,
9 there's no functor b in the database
10 so it looks to the rule b(X), rule number 2,

This subject was relatively unscathed at this point by the database
bug, but he showed the RBFL error in his protocol, and jumping
around in the program this way may have been a contributing factor.

But there were other consequences of forcing a division between
facts and rules which resulted in subjects not properly reading the
whole program - students 'jumped' over whole sections of program as
they moved between the 'database' and the rules. Subject 5
illustrates the case. He began with a very high level description
of what he thought likely to happen, perhaps being uncertain of what the task was:

Subject 5

1 It'll take a(X) here uhm it will try to
2 find conditions in... I suppose the body of the
3 clause here that will satisfy a(X).
4 So it will look first at b(X) and see if it
5 can instantiate or find anything such that
6 er.. b(X) is satisfied and then the same with c(X).
7 Now you've got a variable there X
8 and the same variable here...uhm...
9 I don't know I would have to try it out
10 to see if it worked.
11 I assume it would, I don't know.

The experimenter pointed out that he needed to be more explicit about the execution, and he proceeded as follows, exhibiting uncertainty about the status of variables and constants:

12 Well it will search down the database
13 to see if it can find any instance of X.
14 I don't know if this X is meant to be uhm
15 a variable or a constant...

EXP: it's a variable

16 a variable uhm such that b(X) occurs.
17 Now this is our database here.
18 I think it's gonna probably going to fail
19 so that won't be satisfied
   [indic. b(X) in Clause 1]
20 so I don't think it will ever bother
21 with the second one
22 as that one hasn't been satisfied
23 so therefore it will return 'no'.

This subject had prematurely jumped to the conclusion that the program would fail, but it was not immediately clear at this stage why he should have done so. The experimenter pointed out that he had not picked up any instantiations at which point he commented:

24 Right. Are these rules in as well?
   [indic. Clause 2 and Clause 3]

He had skipped two of the rules in the program an attempt to get an
instantiation for X from the database alone. His explanation was not only incomplete, but also incorrect, because the possible instantiations from the facts in this program were obvious.

Similarly, Subject 8's confusion arose from a combination of being uncertain of what matched with what, and the database bug. He first tries to find a direct match for a(X):

Subject 8

1 Right well, the first...
2 I would look at the first rule
3 and er say that uhmm it [mumble]
4 uhmm so the first rule says a(X) if b(X) and c(X)
5 c of X, so -
6 [pause]
7 that doesn't seem to.....
8 well nothing in the database will match that....
9 nothing in the rules...er facts list
10 seems to match with that...

so he turns to the rules to work it out:

11 so it goes on to the rules.
12 b(X) if d(X) and e(X) -
13 nothing matches there either.
14 Oh! In fact that doesn't work. Uhm...
15 In fact only rule 1 applies really...
16 only rule 1 has a(X) on the left hand side.

His reading of the program precludes the use of the rule Clause 2 - he wanted the sub-goals in Clause 1 to be immediately instantiated from the facts in the 'database'. At line 15 he seemed to have forgotten that Clause 2 must be used to obtain a solution for b(X) which then would be used in Clause 1 to obtain the value for X in a(X). This could be a serious misconstrual since in effect he was ignoring the deductive properties which are the lynch pin of Prolog programs.
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).
```

Fig. 7.6 Modified backtracking task

In this case, a student suffering from TOAP will feel no effect of the cut on execution, since once the clause containing the cut has failed for the first time, then execution will pass to the next clause anyway - no backtracking will take place, so the cut will not be used. In the RBFL, the cut may similarly be rendered useless since although students frequently know that during forward execution, the cut is automatically satisfied, but that one cannot backtrack through it, the RBFL error means that 're-doing' 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.

The effects we were expecting to observe in this modified test were limited by the fact that we drew from the same pool of students, who had been carefully debriefed after the first experiment. The results indicate that students had

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect</td>
<td>5</td>
</tr>
<tr>
<td>No. of TOAP</td>
<td>1</td>
</tr>
<tr>
<td>No. of RBFL</td>
<td>1</td>
</tr>
<tr>
<td>Other</td>
<td>1</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>8</strong></td>
</tr>
</tbody>
</table>

Fig. 7.7 Results of modified backtracking task

learned from their previous test, and also suggest therefore that TOAP and RBFL are not persistent errors. On the other hand, students did have these particular mistakes brought to their attention in a one-to-one situation with the experimenter after the previous test, which is fairly intensive coaching. However, it turned out that the two subjects showing TOAP and RBFL in this test had neither misconception in the previous test.

**Try Once and Pass and Cut**

Subject 5 showed how an apparent TOAP misconception can be caused by the cut. He began his interpretation and at lines 7 and 8, he had correctly identified the fact that the value of X must remain 1 in that clause. However, line 10 indicates that he thought the cut would 'initiate' backtracking:
Subject 5

1 Right, it would take \( a(X) \) and try and see
2 if it could satisfy this first rule here, \( a(X) \),
3 and it would go to the first rule \( b(X) \).
4 um... and try and satisfy that,
5 it would see \( d(X) \),
6 \( X \) could then be instantiated as 1,
7 um... it would then er cut... um
8 so it would then stop with that value 1
9 I think that's right,
10 Um... I'm not sure presumably that initiates backtracking

However, this did not affect the flow of his description, and he went on to get the binding of 1 for \( X \). Interestingly, the binding for \( X \) in \( c(X) \) - i.e. in Clause 1 - was assumed, and was described as being for the sake of consistency (line 17):

14 so that's now satisfied, \( d(1) \), \( e(1) \)
15 so come back to ...
16 \( b(X) \) is \( b(1) \) ... and er
17 for consistency's sake this \( c \) is going to have to be 1
18 so its going to see if it can satisfy \( c \) anyway.

Having tried the instantiation of \( X \) to 1, the student then backtracked out of Clause 2, and tried Clause 3:

24 right, so it'll backtrack to \( b(X) \)
25 and see if there's any other
26 ways that it can resatisfy \( b(X) \)
27 which it can do, matching against this rule here
    [indic. Clause 3]
28 and sees \( f(X) \) and \( f \) is 3.
29 Now, er, yup... ok, so um,
30 takes up the value 3, um
31 so we're potentially \( a(3) \), \( b(3) \), and \( c(3) \)
32 and it finds \( c \) as the first one here,
33 so we're all right,
34 so we've now got um
35 \( X \) instantiated as 3 throughout
36 it'll return \( X \) equals..
37 sorry a equals 3... or \( X \) equals 3...

This student did not have the TOAP problem in his first protocol, and it seems likely that it has arisen here because he saw the effect of the cut as local to a clause. That is, he knew that the only instantiation for \( X \) was 1, and that could not be changed, but
he had missed the fact that Clause 3 should also be frozen out by the cut. In this case, he appeared to have TOAP, because of the supposed effect of cut. In fact it is more probable that he had misunderstood the scope of the cut.

Redo Body from the Left and Cut

Similarly, Subject 6 did not have TOAP or RBFL in her first protocol, though she did have other errors which will be discussed later. However, in this task, RBFL appeared. She began well:

1 Ok, uhm, first thing is er
2 look for a rule mentioning a(X)
3 and find one here  
   [indic. 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(1), 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  
   [indic. 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(1), 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.

Since the cut is essentially being ignored, as she failed with
d(2), she went on to look for another way of satisfying b(X), and doing so led her to try Clause 3 (lines 63-64) which should have been frozen by the cut:

62 There's no other way of trying this rule.
63 There's another rule I can try, uhm this one..
   [ind. Clause 3]
64 Uhm, told me to go and find an f(X),
65 look for one,
66 find one,
67 Now X is 3, uhm...
68 go back to here
   [indic. Clause 1]
69 X is 3,
70 look for c(X) with X as 3.
71 So down my list looking for C,
72 find it here
73 c(3), both of these are satisfied,
74 so I've satisfied a(X) when X is 3.

She was aware, however, that something was not quite right:

75 But I think I did something wrong
76 because I didn't do....
77 there's something I didn't do!
   [indic. !]
78 With the cut there!

In subsequent discussion, this subject emphasised the fact that she did know that one never backtracked through cut. Because of her left to right reading, though, cut was not being 'activated', and therefore had no other effects on the rest of the program. In this respect, her difficulty was not necessarily associated with understanding the cut itself, but with conditions under which it plays a part in program execution.

'Parallel' Execution

One subject had an interesting bug which was exhibited in both his protocols for the two tests. It seemed to be a form of parallel execution. In the first task (the uncut program) he moved correctly through execution, having got the instantiation 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 in this example was avoided, and he continued more or less correctly. In his second protocol, where the cut was included he set off again to prove the 'e' goal, but showed the multiple 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
21 alternative solution for d
22 so it goes back out.
23 Its now got a proof for b(X)
24 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 28), but whether he had the correct reasoning was unclear. It also appeared
from what he said, that the only reason Prolog did not try all these options at once was because of the cut (line 20). The next step might indicate the source of error. There were two c clauses in the program, which he thought would be 'compared' with c(1) for a match (lines 35 and 37):

30 Having proven b(X) it goes on to prove c(X)
31 so we have one solution for b(X) which is b(1).
32 Going on then to prove c(X)
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 and 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(1), so there 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.

Failure to 'retry'

The last, and most common, omission which all but one subject had was the attempt to resatisfy e(1) prior to backtracking into d(1). In one sense this is a minor slip, since in this program e(1) 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(1) 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'. 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.

All of the bugs so far illustrated are examples of quite specific misconceptions within the mechanistic level of discourse. However, an issue which has not been considered is their origin. Coombs and Stell's major concern has been to construct debugging facilities which assist students once these errors have been made. But it has been pointed out that errors are rarely exhibited in isolation from other related misperceptions. The database bug affects the way in which variables are perceived; it can also prevent students from properly reading the program; the cut can induce both TOAP and RBFL in students who did not exhibit these errors previously and so on.

Furthermore, throughout the protocol sessions it was clear that students often became aware that something was going wrong with their explanations, or they felt that they really did know something, but it was not coming out right somehow. Also, some students obviously did know certain facts - e.g. Subject 6 knew that one never backtracked through the cut - but the explanation which was offered did not allow that piece of information to become incorporated. There seemed to be a sense of 'weaving a story' during the session rather than simply reporting known facts. As was the case for the Computers and Thought students, the final product of the session was of less interest than the route which was taken.
Whether their eventual story was correct or not, the process of constructing a plausible account, in which as much concrete and well-known information was included as possible, seemed similar for all the students. In several cases students expressed dissatisfaction with their own final story. This implied that a meta-level perspective was being used to evaluate the story-so-far and to inwardly comment on its plausibility. This in turn seemed to indicate that students would provide different stories at different times, rather than consistently report the activities of some well-established mental model of the Prolog interpreter.

Until now the bugs have been reported as ones which could be described as students simply misunderstanding aspects of the mechanistic view of Prolog. However, as was mentioned earlier, there was a further category of very high level misunderstandings which cannot be viewed this way. In the next section, broader issues in relation to the same set of protocols obtained in these two experiments, and hypotheses are made about possible sources of error.

7.3 Section 2: Meta-level Reasoning Superstrategy

The main argument of this section is that students were using meta-level reasoning strategies to interpret the task, and to make up sensible stories. The information which was used to construct the stories came from several sources and varied from individual to individual, but the overall strategy was similar.

Meta-level reasoning is a very powerful technique which provides a means by which to grasp the general pattern of execution without getting lost in the details. So students were sometimes able to
infer some overall behaviour in the execution domain using either real-world knowledge, or knowledge about formal domains. Upon occasion, they were correct, even though they could not give a blow-by-blow account of the full execution process (a tedious exercise in any case!).

Figure 7.8 illustrates the view of the task adopted in this part of the discussion. The input is, as before, a program, so we begin at the mechanistic level. But the student now begins to move up to the general level to reason, and this is initially an intuitive move (indicated by a wavy line).

<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>REASONING</td>
</tr>
<tr>
<td>INTERP.</td>
<td>STRATEGIES</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>REFLECTIVE</td>
</tr>
<tr>
<td>EVALUATION</td>
<td></td>
</tr>
<tr>
<td>GENERAL</td>
<td>Implicit</td>
</tr>
<tr>
<td></td>
<td>Ordinary</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
</tr>
<tr>
<td></td>
<td>Empirical</td>
</tr>
<tr>
<td>MECHANISTIC</td>
<td>Explicit Analytic</td>
</tr>
<tr>
<td></td>
<td>Explicit Hypothetical/</td>
</tr>
<tr>
<td></td>
<td>Empirical</td>
</tr>
</tbody>
</table>

Fig. 7.8 Ideal Pathway for Coombs and Stell Test

Ideally, the student would then map downwards to the mechanistic domain to take into account information at that level, produce output which was explicitly formulated, and hypothetically evaluated. The important point is that however the student moves up to the general level, she must map downwards at some point into the mechanistic domain. The language which the student uses should change as this shift takes place.

To illustrate, Subject 4 begins with meta-level reasoning, but then switches to mentally executing the program. He started off talking
about what he thought the program would do:

3 Clause 1 - a(X) if b(X) and c(X),
4 uhm I think I'd probably start
5 and look down at the facts
6 to see what might be instantiated.

and after having looked at the facts he noted that c can only be
7 instantiated by the fact c(3):
8 Well c is going to be instantiated by that fact c(3).
9
But he then went on to consider how Prolog would be able to
10 establish that instantiation. During consideration of Clause 2, he
began hypothesising:

8 Clause 2 b(X) if d(X) and e(X)
9 uhm and are both d and e are going to
10 be instantiated
11 and d will be instantiated by this one first.
12 Uhm and 3 b(X) if f(X) and f(X) is instantiated.
13 Uhm I'm getting a feeling that if it's asked
14 to backtrack
15 it will get the second ones for b for d for e
16 but if not it will just give one result.
17 Now what's the result going to be?

At this point he changed tactics and began to interpret the program
according to Prolog's execution strategies:

16 Well its going to set up sub-
17 ...suppose we give it the goal ?-a(X)
18 it'll set up the subgoal b(X)
19 and in trying to satisfy that it will take
20 clause 2 here
21 and then try and satisfy another subgoal d(X)....

The shift is indicated in lines 17-18. In effect, the student's
point of view has changed from 'me doing it' to 'Prolog doing it'.

This is a clear example of unconfused meta-level reasoning.
However, the meta-level strategy can become troublesome when the
student makes assumptions about the execution domain on the basis
of prior knowledge, or experience with another, perhaps similar,
task. 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 errors 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.

As was pointed out in Chapter 5, the program used in the studies has a flaw in it because it actually encourages meta-level reasoning to take place in a way which allows subjects to avoid doing the task. Subject 1 illustrates this process. His first comment was:

1 Well f will instantiate X to be 3
2 so b will be instantiated to be 3

The experimenter encouraged him to work through the program, but instead of considering the execution path, he still would only work from the variable instantiations:

30 The only way you can get b instantiated is from this, [indic. Clause 8]
31 so it would be 3,
32 in which case d would have to be 3,
33 and you haven't got a d being 3
34 you've got no e being 3
35 and that one would fail
36 in which case X could only be 3..

In his verbal report, therefore, he omitted to discuss the backtracking to get the successive instantiations of 1 and 2,
before Clause 3 is used. However, it seemed likely that this was a case of misinterpreting the task, helped along by the test program, rather than being muddled about the execution. He seems to have interpreted the task as 'what do you think this program will do?', rather than 'what do you think the computer will do with this program?'.

Having shown the use of the meta-level reasoning superstrategy, we now move on to discuss superbugs which can arise, which are listed in Figure 7.9 for reference.

* Meta-level Reasoning Superstrategy
  * Identity Superbug
  * Wishful Thinking Superbug
  * Left-to-Right Bias Superbug

Fig. 7.9 Meta-level Reasoning Superstrategy and Superbugs

The Identity Superbug

In the two cases described above, both students had kept separate their own knowledge of what they thought might happen, and what they thought Prolog's execution strategy would be. The following protocol illustrates a case where the student, began to slide from this initial view into a confusion between her own strategies, and those involved in execution. This has been dubbed the 'identity superbug' because the distinction between 'what I would do' and 'what Prolog would do' becomes blurred.

Subject 6 was already confused about the relationship between the logical interpretation of programs which she had been given in lectures, and Prolog's execution strategy. The identity bug appeared as she tried to reconcile the two. This subject had quite
a lot of knowledge both about the process of resolution and about execution. But during the process of amalgamating logic and execution, her own reasoning strategies became jumbled with Prolog's activities, which led her to apparently attribute to Prolog the sort of goal-directedness, or foresightedness identified by Pea (1986) in intentionality bugs.

She 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 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.
EXP: Right, ok. So a(X) to set you off....

12 Right., so now I've got a(X) I'm a(3) if b(3) and c(3)....

In other words, she still had the spurious instantiation of 3 for X. She began to realise something was wrong, though, as she continued:

13 .....Ahm...Ok now what I'm hoping
[indic. Clause 2]  
14 is that - this is getting a bit confused now...
15 Now I'm not Prolog any more, but me being a little confused because
17 I'm hoping that I can also instantiate b to the same thing. what I just instantiated c to,
19 so that....so that I'm justified - I think I'm confused now, 'cos I'm doing it half in Horn clauses and half just in my head, saying what needs to be the case for 'a' to be the case.
23 And I kinda set about doing it as if I were resolving it and only doing half cos I'm not adding the negation of what I'm trying to prove.

The experimenter was still concerned that she had an 'illegal' instantiation of 3 for X, and the subject confirmed in her subsequent comments that she was confusing her own strategies with those of Prolog:

EXP: Right. What I'm interested in is why you have picked up an instantiation - or at least how you've picked up the instantiation of 3 for X.

26 Being myself?

EXP: Well that's how I think you did it!

27 It is isn't it! What I did was in just sort of English, not really in logic - I was trying to find the same value ahm of which its true that both b of that value and c of that value, so since my first rule
[indic. Clause 1]  
32 mentions b and c there was no b immediately available to me
[indic. facts]  
34 so I thought Oh, ok I'll go and investigate that later - maybe I'll have to
[indic. Clause 2]
36 use this rule, or this one
   [indic. Clause 2 and Clause 3]
37 but there is a 'c', I can get my 'c' to be
38 true immediately and I'll be able to get 'a'
39 - I'll be able to use a(3) if I've got c(3)
40 and then kind of on my agenda of things to be
41 done, I can also establish b(3).

The fact that her 'agenda of things to be done' did not correspond
with Prolog's agenda further confirmed a confused understanding of
the execution level. The experimenter tried to stress that it was
the mechanisms which allowed instantiations to take place which was
important:

   EXP: What's the mechanism that would allow Prolog
to pick up any instantiation at all for X,
and what Prolog wouldn't be able to do is
jump to the end of there and scan the
database. It wouldn't do it that way. Now can
you tell me how it would do it?

Getting confused with the resolution process, the subject realized
that she did not really know:

42 Uhm...I think how it would do it is ..... 43 well I suppose how it would do it...
44 All its got to play with are these facts
45 and a new fact which its trying to say not(a(X)).
   [i.e. the query]
46 Ands it's going to try and.....no.
47 Now it's apparent to me that
48 I don't really understand how its going to do it.
49 Cos what I want to say is its going to try
50 and resolve the new fact with what its got
51 but it's also obvious that the new fact doesn't,
52 first off, resolve with anything.

Line 52 illustrates a problem which cropped up for several other
students - because nothing matched with instantiations immediately,
they were unclear as to how Prolog 'got started'.

The experimenter encouraged her to think only of the execution
rather than logical resolution, and the subject began her
explanation, demonstrating that she did in fact know a great deal
about what would happen:

54 I see that it goes here to No. 2
55 because it's the first clause that mentions of \( b(X) \)
56 on this side,
57 or just as a fact on its own.
58 So then I would say that it looks at this rule
59 and says \( b(X) \) is true if these two things are true
60 so... it adds those 2 things
61 and the first of them to be addressed is \( d(X) \).
62 First \( d \) it comes to is this one
   [indic. Clause 4]
63 so it tries... it tries instantiating \( X \) to 1,
64 puts 1 in for \( X \) here
65 then I would say backs up the goal queue
66 so the next thing it has to worry about
67 is whether that's a valid instantiation,
68 also here
   [indic. \( e(X) \) in Clause 2]
69 since it had taken off \( d \)
70 and the next thing under was \( e \).
71 Looks down for some \( e \)'s.
72 Here's an \( e \) - its 1, so far so good.

EXP: Why is that good?

73 Because in order for \( b(X) \) to be true
74 both \( d \) and \( e \) have to be true of the same \( X \)

For example, she knew that search in Prolog is exhaustive - Clause 2 in this program failed on two occasions, and she emphasised that it would not terminate until all possibilities had been tried.

She then highlighted a problem in the test situation. The instructions to students were to describe out loud how Prolog would interpret the program. She pulled herself up at one point, and commented:

112 Why I stopped myself is 'cos I just did that in my head
113 as being kind of obvious,
114 but I suppose it's not obvious
115 it is actually a step.

She was referring to the fact that prior to backtracking into the 'd' goal in Clause 2, the possibility of resatisfying \( e(1) \) must be checked:
Right. Why I hesitated is 'cos I realised I had done it in my head for a slightly wrong reason which was sort of what was in my mind was Oh, X is 1 now so I was scanning through looking for another e expression involving X is 1 and that is in fact the criterion by which to work out in the end for my purposes but I think its the wrong kind of orientation.

This student was the only one who even mentioned this aspect of the execution. She then produced the following personification of Prolog:

I think being Prolog first he'd look for an e and then work out if he can get an X is 1 whereas I just said instantly oh there's an e but it's 2 so I won't even mention it to you.

It is clear that this subject knew quite a lot about how Prolog operates at the lower level, but had initially confused herself by trying to incorporate the logical perspective into her explanation. Only at the insistence of the experimenter did she stay at the execution level, whereupon she produced a perfectly competent account of the process.

This protocol provides an illustration of how a student initially tried to use information and concepts from all three levels of the discourse framework to explain a program's execution, which resulted in a muddle (see Figure 7.10).
In other words, she began by moving intuitively from the mechanistic domain up to general reasoning strategies during which time she was confusing her own strategies with those available to Prolog. She then tried to present a logical interpretation of the program, but became aware that this was going to be inadequate partly because she had an incomplete grasp of the underlying logic, but, more importantly, she began to recognise that this level of explanation (i.e. using resolution) could not provide her with information she needed because she couldn't work out how it would get started (cf lines 49-52). Eventually she recognised that a mechanistic explanation was what was required and she proved capable of providing it in some detail.

This situation is very marked in Subject 6's protocol, and some of her difficulties may be artefactual due to the use of verbal data - i.e. if she had to write down what happened, or construct a program herself, this kind of misinterpretation might have been less evident.
Nevertheless, an effect of the identity bug is the assumption that Prolog can move around the program in much the same way that the student can. Further instances can be found of this misapprehension, and appear to fit into the notion of 'wishful thinking' identified by Elshout et al. (1986).

Wishful Thinking Superbug

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 - i.e. this might have been the case in Subject 1's protocol, but his subsequent comments were consistent with knowledge of the execution.

But, for example, Subject 10 had a difficulty which indicated that she had an incomplete interpretation of Prolog programs and left the rest to 'wishful thinking'. She appeared only to find an instantiation for the first sub-goal in every clause. Having started off with b(X), and upon finding d(X), she returned to the parent goal with an instantiation of 1:

Subject 10

1 Look for a(X),
2 so it has to satisfy these goals
   [indic. Clause 1]
3 taking b(X)...
4 uhm so it must go to here
   [indic. Clause 2]
5 and this depends on ... if d(X),
6 so it looks for an instance of d(X)
7 so it instantiates it to 1
   [indic. Clause 4]
8 so that's instantiated to 1
   [indic. b(X) in Clause 2]
9 that's instantiated to 1
[indic. b(X) in Clause 1]
10 and that's instantiated to 1
[indic. a(X) in Clause 1 and the query]
11 So X is 1.
12 Then it waits for you to ask again.

Up until this point she had only discussed instantiations for b(X), and had ignored e(X) in Clause 2 and c(X) in Clause 1. The experimenter tried to point this out but was interrupted by the student, who then revealed her own assumption - that Prolog 'presumes' that c(1) is true because b(1) is true (line 16):

13 Oh yeah, sorry I haven't finished,
14 and it goes on....
[indic. c(X) in Clause 1]
15 I'm not sure whether it then
16 presumes that c is 1 as well
17 if you're, you know...to finish it.
18 I would have presumed it would
19 once it had found that
[i.e. that X is 1]
20 then it would.....
[pause]

EXP: And then it would finish?
21 Mmm.

After the experimenter pointed out the omission, she commented:

22 Sorry I completely left out e didn't I!

and she tried again:

23 Well....I would have just gone on
24 and said that goal is satisfied
[i.e. d(1) in Clause 2]
25 and e...it picks up e,
26 the first case of e(1)
27 and then goes back
28 satisfying all its goals up there.
29 Uhm...then it would stop
30 and wait for you to ask again.

EXP: Ok, so you got d(1) and then you checked
31 that you'd got e(1) to come out here
[i.e. out of Clause 2]
32 so you've proved b

31 Oh yeah...
... with X instantiated to 1.
So what happens next?

32 Oh that's what I wasn't sure
33 whether it would actually just accept
34 that b(X) is..was X
35 and be happy
36 or whether it would actually go through again
37 looking for .....:

She still had the problem of not checking for c(l) in Clause 1. The experimenter tried to probe why, if the sub-goals d(X) and e(X) were resolved, that c(1) would not similarly be dealt with, and the subject further revealed her unsureness:

EXP: Right. What's the difference between this clause
[indic. Clause 2]
and that clause
[indic. Clause 1]

38 Uhm. In what way?
EXP: In their structure...

39 Uuhmm....

EXP: Because when we did this one
[indic. Clause 2]
we found an instantiation for d, and then
checked there was one for e. Why wouldn't we
do that here?
[indic. Clause 1]

40 Well, I wasn't even sure there
[indic. Clause 2]
41 whether in fact it would actually do that or not.
42 So in fact it does actually, it doesn't just
43 take it for granted?
[indic. e(X) in Clause 2]

The student hoped that Prolog would just accept instantiations, or 'take them for granted' rather than have to prove the remaining sub-goals. The difficulty of knowing what can and cannot be left to the machine to perform automatically was discussed in Chapter 4, and it can be problematic for students to establish the boundaries precisely when some aspects of the machine's behaviour are apparently 'magical' (e.g. backtracking and unification).
On the other hand, 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. These are examples again of meta-level reasoning being used to perform the task.

For example in Subject 8's protocol:

Subject 8

17 Uh huh. Yes, right... ok..... ahhh...
18 Yes well, uhm... so it goes to rule number 2
19 and er and it finds b(X) if d(X) and e(X)... 
20 uhm... - pause - and uhm [cough] and uhm 
21 I don't think it'll get very far with that. 
22 So then it goes on and it takes the second rule,  
23 b(X) if f(X).

Line 21 indicates that he was uncertain how to interpret Clause 2, and so he passed straight on to the slightly easier Clause 3.

Of course, once students begin jumping around in programs rather than adopting the rather myopic view required to simulate execution, the scene is set for many kinds of misapprehensions. The database bug discussed in the above section can be seen as just another, rather concrete, example of how introducing prior knowledge and assumptions into the task affects interpretation in this respect. The point is though, that the database bug might be expected if it is known that students have been learning another programming language. However, bugs arising from superbugs do not have so obvious an origin, and are less easy to classify. Furthermore, since superbugs tend not to be acknowledged, teaching materials and teaching techniques can inadvertently cultivate them in the ways described in Chapter 5.
Another superbug which appeared to influence students is one where a left-to-right interpretation is favoured. 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. 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 execution does progress from left to right. But the tacit (and potentially confusing) assumption is that the flow of control through a clause or through programs is always left to right.

This assumption 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.

In order to understand the sorts of difficulties this superbug can cause, it is necessary to discuss an example of how Prolog clauses can be used which will then be used to illustrate 'left-to-right bias' superbugs. The clauses are those used to define the 'append' relation. These are examples of particularly succinct Prolog clauses, and beginners would not normally be expected to come to grips with them immediately. But they do nicely illustrate certain important facets of Prolog's behaviour. The standard definition of
the 'append' relation is as follows:

Clause 1: append([], L, L).
Clause 2: append([H|L1], L2, [H|L3]) :-
        append(L1, L2, L3).

where the first two arguments are typically input variables, and
the third is an output variable. An interpretation might be:

append some list TO another list TO PRODUCE a third list.
(Clause 1 deals with the case where the first list is empty, and
Clause 2 deals with the case where it is not empty.)

However, this reading of the 'append' clause is by no means always
the case. The clauses can take instantiations from the input query
for any of these arguments. The resulting behaviour will then be
radically different. For example, exactly the same clauses can be
used to break up lists (on forced backtracking):

Example 1: ?- append(X, Y, [a,b,c,d]).

In this case, the list [a,b,c,d] will be broken down as follows:

X = []
Y = [a, b, c, d] ? ;
X = [a]
Y = [b, c, d] ? ;
X = [a, b]
Y = [c, d] ? ;
X = [a, b, c]
Y = [d] ? ;
X = [a, b, c, d]
Y = [] ? ;

no
(The semi-colon indicates that the programmer has forced
backtracking.)

This is frequently known as using clauses 'backwards' - i.e. what
is usually the output variable is used as an input variable. Example 2 illustrates the case where Prolog will match a pattern in a list (taken from Bratko, 1986, p.70):

Example 2: ?- append(Before, [mar|After], [jan,feb,mar,apr,may]).

As 'append' runs, all months before March will be put into the first argument, and months after March in the tail of the second argument, in the following way:

Before = [jan, feb]
After = [apr, may]

Append can also be used to delete elements from lists:

Example 3: ?- append([a,b,c], X, [a,b,c,d,e,f]).

will produce the output:

X = [d,e,f]

In order to understand what function clauses are meant to perform, then, it is crucial to know what variables will be instantiated in the input. A good habit to encourage is the insertion of a clause in comments to a program indicating the calling mode. For example, the calling mode for the usual use of append would be:

append(+, +, -).

where '+' indicates an instantiated variable, and '-' indicates an uninstantiated variable. The 'backwards' use of append would be signalled by:

append(-, -, +).

Understandably, beginners often need quite a while to come to understand 'append' properly. The example illustrates clearly the need for appreciating precisely how unification works.

The first manifestation of left-to-right bias is the tacit
assumption that instantiations in clauses always pass 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 different behaviour. Assuming that the student in the early stages of learning Prolog is not yet able to fully understand a complete description of how 'append' works, she may well be confused to see a clause performing different functions apparently without itself being altered in any way. The idea that when 'append' has its rightmost argument instantiated, it is probably better named 'split', but that the clause structures need not change is somewhat counter-intuitive. In other words, the word 'append' is only 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 usual way, then she may be justified in thinking that the word 'append' now means (to the computer) what the structures in the clauses define it to mean, and that this corresponds to what she understands it to mean (i.e. joining lists together).

This can combine with what might appear to be natural language bugs - when students attribute 'meaning' to predicate names it is not a
simple case of thinking that Prolog 'understands' 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 the student thinks is defined as a 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.

A second manifestation of the left-to-right bias superbug is attributing 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:

\[
\text{append([], L, L).}
\]

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 two arguments L and L must be able to unify.

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 each level of the recursion, the first argument will be represented by an empty list (because Prolog is moving down the list), and (b)
that at some point the goal which matches the clause head will be:

```
append([], [mar|Rest], [mar, apr, may]).
```

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.

This kind of error may be associated with errors noted in logical inference rules, where the expressions are parsed with some notion of 'primary' and 'secondary' clauses - i.e the first encountered clause reading from left to right is salient, and the second clause following is somehow dependent. In Modus Ponens the argument form follows a 'sensible' pattern according to this schema:

```
p implies q
p
------
therefore q
```

where the secondary clause hinges on the primary - the primary is given, therefore the secondary exists also. However, Modus Tollens goes against this schema:

```
p implies q
not-q
------
therefore not-p
```

where the secondary clause affects the primary. Fewer people are inclined to accept Modus Tollens as valid from an intuitive point of view. The fallacy known as denying the antecedent is commonly thought to be valid:
\[
p \implies q \\
\text{not-}p \\
\hline \\
\text{Therefore not-}q
\]

The fallacy of confirming the consequent, however, which is equivalent to the kind of abductive process described earlier, goes against the schema:

\[
p \implies q \\
q \\
\hline \\
\text{Therefore } p
\]

Reading left-to-right might therefore be a default assumption, unless the beginner has a vested interest in doing otherwise!

7.4 Discussion

A two-levelled view has been taken of MSc students interpreting a program designed to investigate backtracking misconceptions. The first part of the discussion focused on the mechanistic domain, where the emphasis was on concrete bugs. From this perspective, the students performing the task of interpreting the program were shown to have several bugs in their interpretation. 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 second part of the discussion which focused on superbug interference identified other bugs which, if only a mechanistic level view was taken of the task, would 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 mechanistic discourse. For example, 'try once and pass' and 'redo body from the left' could themselves be interpreted as examples of the left-to-right bias superbug. The database bug was included in earlier discussion because it seemed to be a case of interference from other programming concepts. That is, rather than percolating down from a general discourse level, this misapprehension was an instance of overlap from the mechanistic domain associated with POP-11. But it could equally be described as prior knowledge affecting interpretation, where meta-level reasoning dictated that existing knowledge of POP-11 would be useful in the current task. The net result is a disorientation during execution, a difficulty which can arise from several sources. This issue will again be taken up in the case study in Chapter 9.

It might be argued that one way to avoid such misapprehensions is to provide students with 'clean' virtual machines to pre-empt the sorts of superbugs described here. However, referring back to the arguments presented in Chapter 2, and tying in the observations reported in this chapter, it is evident that students do not simply imbibe correct information and regurgitate it upon request. Instead, information from many sources is used to construct explanations 'on the fly' at this stage in learning. As was suggested in Chapter 4, learning may be a process of successive approximation towards a personalised version of a correct model which can only be constructed by the learner (i.e. it cannot be implanted) and will only become stable when it is able to help the learner generate explanations to her own satisfaction. When this ability has been developed, in effect, the learner will have moved
on from novicedom toward a state of expertise.

7.5 Conclusions

There are two major points arising from the study. Firstly, whilst it is useful to identify the particular manifestations of errors, it is equally important to keep in mind the strategies which beginners are using. 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 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 not only be fully exploited, but also appropriately discarded. This point has been made by Bonar and Soloway (1982) who discuss how students often come to learn more
conventional instructional languages with 'healthy' and powerful intuitions about step-by-step specification of solutions in natural language. They argue that the cause of buggy programs lies in the fact that most programming languages are not designed to appeal to these intuitions.

Many of these bugs bear resemblance to Pea's (1986) 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' or stories about what 'should' happen, and, as has been discussed earlier, these sorts of abductions may be subjected by learners to verification rather than falsification (cf the discussion in Chapter 3). 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.
The next chapter provides an illustration of such an abduction, and demonstrates the persistence with which a student can stick to an original hypothesis in the face of opposition from another student.
CHAPTER 8

Abductive Reasoning

8.1 Introduction

This chapter illustrates the powerful effect of abduction in reasoning. Recall that three pairs of undergraduate Computers and Thought students were presented with the standard clauses which define 'membership', but with the meaningful predicate name taken out (see Figure 8.1).

\[
\text{foobaz}(X, [X|_]). \\
\text{foobaz}(X, [_|Y]) :- \text{foobaz}(X,Y).
\]

Fig. 8.1 Interpretation Task for Computers and Thought

This exercise was conducted seven weeks into the Prolog course. They had recently had lectures on list processing and had used 'member' in a route finding program. However, it was not known prior to the session whether they had seen it as represented above, or whether they had simply used it as part of a teaching package. It turned out that students had not yet been introduced to the underscore variable (_).

The original aim of the exercise was to find out how much information they could extract from the syntax of these clauses with the knowledge they had so far. For example, could they ascertain that the clauses were recursive, that some list processing was taking place, that there was a matching operation in the first clause and so on. This was a very open-ended exercise and no prior hypotheses had been generated.

As pointed out in Chapter 5, unfortunately, the protocols from two
pairs turned out to be of little value, since social factors in the experimental situation obscured the data. One protocol, however, illustrates an example of abductive reasoning. The two participants are Julie and Mary.

8.2 Abduction and the Discourse Framework

Recall that abduction is defined as adopting some hypothesis as an assumption if, together with beliefs already held, some observation can then be accounted for. Having adopted the assumption, the student now has a 'space of discourse' which contains the new element, and it is this space which forms the material to be reasoned about. Whilst abductive reasoning is a powerful learning method, if applied inappropriately, it can lead to difficulties because novices may introduce a spurious assumption which then they try to justify or verify. This can in turn blind the learner to other information which suggests that the hypothesis was incorrect.

Abduction is an important technique by which strategies and information from other domains can be integrated into the novice's perception of an unfamiliar domain. This process, of course, affects the learner's perception of the new domain, and it may be that having made one mistaken abduction, whole series of other abductions have to be made in order to keep some consistency in the learner's interpretation of the domain. On the other hand, if abductive reasoning were not performed, most learners would never get started, so it is an essential component in learning.

The protocol discussed here shows one example of how the inscrutable Prolog clauses above led one participant to make an abductive hypothesis about syntax. The significant point, though,
is not that she made it, but the vigour with which it is defended in the face of opposition from the other participant, and direct information from the experimenter. This kind of defence has ramifications for teaching methods, and analysis of performance since it demonstrates that simple presentation of 'correct' information may be insufficient to overcome powerful intuitive interpretations. Figure 8.2 illustrates the particular pathway in which the abduction is made from the mechanistic input to general reasoning strategies.

<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>INPUT</td>
</tr>
<tr>
<td></td>
<td>INTERP.</td>
</tr>
<tr>
<td>GENERAL</td>
<td>Implicit</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Fig. 8.2 Pathway for Abduction in Interpreting Prolog Clauses</td>
<td></td>
</tr>
</tbody>
</table>

8.3 Julie and Mary: A Protocol Study

A fundamental problem for Julie and Mary in this exercise is that they tried to work out what the clauses did without being systematic about substituting objects for variables. They knew that they needed to do this, but were thrown somewhat by the presence of the underscore variable which they had not seen before. The underscore represents a variable, but its use indicates that the programmer is not concerned to find out what the instantiation of the variable is. This saves Prolog from having to check whether the
value fits a particular pattern, which is wasteful if it will not be used. In this case, the first clause tests only the head of the list for a match, and if it is successful, execution will terminate. Therefore, the rest of the list is irrelevant, so the underscore is used in place of a variable name. On the other hand, if the program gets to the second clause, the head will have been tested by the first clause, which must have failed. There is no point, then, in checking the head of the list again. However, none of this can be guessed at from the code, which offers no clues as to the status of any of the objects contained within it.

Given this confusion, Mary adopted an interpretation of the underscore variable as a 'blank' which was reasonable. However, an idea brought to mind by an exercise she had been doing earlier in the week with transitivity links suggested to her that it needed to be filled in. The exercise she recalled was a route-finding program which constructs a transitive chain to get from X to Y by establishing whether there is a link from X to a new place Z, and a link from Z to Y. The program constructs a list which constitutes its 'trail' as it moves around. Mary decided that the program had to find the 'missing portion' between X and Y in the 'member' clauses.

Although in retrospect it is obvious that she was thinking along these lines from quite an early point, she did not make this explicit statement until quite a long way into the protocol. The underscore plays a crucial role in the abductive process, because the perceptual arrangement of clauses encouraged Mary in her assumption that Prolog was 'looking' for something to fill up the gaps. It is as if the underscore at the end of the list in Clause 1
was going to be used to 'plug into' the underscore at the front of the list in Clause 2, as if the code were written using 'difference lists'. Thus, reasoned Mary, X was the beginning of the list - the starting place - and Y was the end - the finishing place. X and Y would also always be instantiated, but anything could go between these two. Having introduced this idea into her 'space of discourse' Mary's job was to describe how Prolog would find the 'missing elements' between the two variables X and Y.

Julie had a great deal of difficulty understanding what Mary was talking about because she herself had the fundamentally correct idea, but found it difficult to articulate. Despite being very close to a correct answer herself on several occasions, she felt obliged to understand Mary's interpretation. But because they were talking about two very different problems - one transitive links, the other membership of lists - they were ultimately unsuccessful in properly interpreting the clauses.

Within the first few minutes, Julie had correctly identified the major characteristics of the clauses - that they were recursive, they took the head of a list, performed some operation, and then recursed down the tail:

1 MARY: do you know what that is?
2 JULIE: it's to do with the head of the list it's that..
3 it's to do with lists and this is
4 this system for taking this
5 first part of the
6 list
8 JULIE: isn't it
9 and then for moving down the list

Basically, Julie had identified the fact that the clauses were list processing clauses, and that the list was going to be dismantled.
This is elaborated by the following explanation:

16 JULIE: it must be the thing for splitting up the list
18 mustn't it it's got to be.
19 That's they need the two you need
20 I'm sh...
21 uuh God, yes, I don't know
22 You need the two clauses you see don't you
23 when you're splitting up the list
24 because they've got to be able to
25 take the head of the list
26 I think (chuckle)
27 and then take the rest of the list
28 right
29 are you with me?
30 MARY: mmmm
31 JULIE: so you do the fi...
32 I think you have a clause first of all
33 to be able to take the first thing in the list
34 and then to take the second thing in the list
35 you go, you have another clause ..... 

She was trying to capture the recursive process in which 'they need the two' (line 19) indicating the two components of the clause, one to do the recursion, the other to act as a stopper. However, during this time, Mary appeared to be musing on the relationship between X and Y. She more or less ignored Julie's interpretation, and stated half way through it:

10 MARY: I know it's connected to something
11 there though and there....
14 X is going to be in the list
15 Y is going to be in the list (mumble)

Mary suggested that the clauses were recursive because the term 'foobaz' was to be found on both the left and the right of Clause 2:

42 MARY: I would have said actually
43 that was the stopping clause
44 so that's.... cos this
45 that is a recursive rule I think
46 'cos that's on the left and that's
47 on the right
and basically, lines 48 - 50 show that she has a partial understanding - she has identified the pattern match in Clause 1: 
foobaz(X, [X|_]).

48 so when it gets to foobaz when
49 it's got X and it's got X in the list
50 that's it it stops

She was confused by the underscore variable though:

51 although I don't know why
   [smile]
52 cos that
53 I've not come across that blank
   [indic. underscore variable]
54 before or the underscore

Julie suggested that it was 'just a space' (line 61). She then looked at Mary, who did nothing, and immediately revoked this interpretation:

62 Oh I see what you mean
63 no we don't know what that means do we?
64 MARY: if it does mean something
65 or if it's just saying
66 that could be anything

At this point Mary commented:

69 MARY: I would expect another variable there
70 a list or a trail at the end here

which is the first indication that she was thinking about her route-finding program. Her interpretation so far was that the first clause was a recursion trap which will stop the recursion when X is found in the list. This is correct, but she was still worried about the underscore variable:

76 MARY: mmm
77 and it's going to keep going
78 round like this
   [circles pencil]
79 this is going to stop it
because when you've got X here
and you've got X in the list
it's going to stop.
This is confusing me because
[indic. underscore variable]
I don't know what it means

Julie appeared to think that Mary had suggested that the first clause itself was recursive as it stood, and was unable to work out what the 'second bit' of the second clause was for (i.e. what is in fact the recursive call):

but can't, I mean,
doesn't that explain to you then what
what that means
I mean I just don't understand why
it's recursive if that if that
the second bit the foobaz X Y if
[indic. the recursive call]
on the... why do we need a second bit on this?
if that's recursive?

At this point, Mary changed her mind, deciding that it was not recursive after all:

mnm
I'm going to take back what I said.
I'm going to say actually I don't
think it is recursive

For no apparent reason, Julie immediately agreed, but asked why Mary had changed her mind. Mary offered as explanation the following rather inconclusive remark:

because it would have some other
something else down here
it would have um something else possibly

Mary still was pushing towards the idea that the clauses were creating a list rather than splitting one up:

I don't understand why it's set up like that
Its going to make up a list of foobaz isn't it?

yup
128 MARY: and lots of ... all the foobazs that it knows
130 JULIE: yep
131 MARY: could fit into the list

This suggests that the abductive hypothesis has introduced the idea of trails and transitivity, made more explicit in the following statement, where the necessary third variable for a transitive relation is introduced (line 137):

134 MARY: so it's going to start
135 it's going to be given
137 a list to make up of foobaz X Y Z
139 do you think? or...

Mary then clarified her interpretation:

175 MARY: but it needs to be more than
177 X Y because we need a list
178 made up with X and Y in it
179 but it might have something in between X and Y.
182 Y is going to be at the end
183 of the list I think
184 and you can have anything between X and Y

but Julie was trying to work out what the variable instantiations in the clauses will be, a point about which Mary was very certain:

186 JULIE: yeah but that variable is going
187 to be instantiated isn't it?
188 I mean those variables

[interrupt]
189 MARY: X Y will always be instantiated
190 JULIE: yeah
191 MARY: um Y is going to be pushed to
192 the end of the list all the time

Julie obligingly tried to convince herself that she knew what Mary was talking about, but was then curious about the X which occurs outside the list brackets (line 201). Mary confirmed her idea that this was the beginning of the list:
264

194 JULIE: but is that the only one that will fit there,
196 that oh I see what you me...
197 yeah I see what you mean .. wait a minute I think
199 I'm beginning to understand what you mean.
201 Well what's this X at the beginning here then?

202 MARY: I think that's the start of a list

another clue as to her line of reasoning, in the suggestion that
they 'want to get from X to Y' (line 212).

203 JULIE: yeah but I just don't understand why
204 it's in that format then.
208 I don't understand this

209 MARY: um .. if you want a list with this X at the front
212 and you want to get from X to Y ....
213 list with X and Y in it

214 JULIE: yeah

215 MARY: so you're going to scan the list of Xs and Ys
217 it's not likely that X and Y will be in...
219 will match straightaway um otherwise
221 there wouldn't be any point in having that list
222 it's going to instantiate X to start with
224 and then it's going to go down and find a Y
226 and Y is going to be pushed .... into the list

At this point Mary decided that they did need to know what the
underscore character meant, and so asked the experimenter (line
239), who offered an explanation:

245 EXP: that underscore means that um
246 you don't care what else there is ..
248 it's the same thing as a variable name
250 but it means
251 don't ever bother trying to find out
252 what the value of this variable is
253 because I don't care

Mary tried her idea out on the experimenter in the hope, perhaps,
of confirmation:

254 MARY: but that's what you're trying
255 to find out
256 by using this rule aren't you
257 you know what X and Y is
258 and you want to find whatever
259 comes between X and Y

The experimenter, without commenting on this interpretation, added
the further information that the underscore can represent either a
single object, or a list, just as an ordinary variable could.

Julie decided, however, to stick with her original idea:

286 JULIE: I mean my instant reaction right would be
287 to say that we are
288 it's looking for two halves of a list
290 or I mean
291 it's looking for the thing that's
292 at the beginning of the list
293 and then it's looking for the
294 rest of the stuff that's in the list

But despite knowing now what the underscore represents, Mary did
not change her interpretation:

295 MARY: I wouldn't
296 have thought
297 JULIE: no?
298 MARY: that it's looking for
299 the beginning
300 because you give it the beginning
301 by giving the X
302 Its looking for what's in
303 between X and Y
304 that's ....
305 that's how I see it

Julie, wisely, tried to push her point further by suggesting they
should think of the inputs to the clauses. However, Mary was unable
to answer any of these questions:

312 JULIE: no, say you got this in your program, right,
313 MARY: mmm
314 JULIE: that's in there,
315 what sort of questions are you going to ask that
316 that's the only way I can think of it
317 MARY: um
JULIE: so you're gonna if you ask it a question

MARY: mm

JULIE: like that

MARY: I would say if you're going to ask it a question you'd be asking it um foobaz X Z

JULIE: Yeah but what do you mean by that? I mean why would you be asking... yes so why... and then what would it do?

MARY: um [grins]

JULIE: yes that's it you see cos that's what I don't under... I mean the way you're talking I don't really understand what the...

[interrupt]

Mary now declared herself, and confessed what has been driving her explanation so far:

MARY: I keep wanting to picture in my mind something like I've already done um [draws] X Y you know these chains [writes X Y as in a ] [transitivity relation]

JULIE: yes, quite, yeah

MARY: and I keep thinking that foobaz X and Y would be here foobaz Z would be here and you'd want to get from X to Z using that

But Julie could not see how that explanation relates to the clauses. By line 484 Julie had, in fact, more or less solved the problem. X is described as a 'number' (line 488) that is being sought for (i.e. X will have a value) and Prolog would check the list to see if X was present (lines 492-494):

JULIE: No, you see, no what I'm thinking I thought that this X here was that...
488 that is the number we're looking for
489 it's the thing we're looking for isn't it
490 that's only there -
491 the X comma is only there to say
492 this is what we are looking for out of this
493 so then you have the list.
497 That's what I think it means
499 then X is what you're looking for out of this list
501 so here you're looking for the X, X

But she was unable to convince Mary at all, who was still contemplating how to build a structure:

508 MARY: You'd have to have something
509 to start off the list though,
510 and I would say X was the start

511 JULIE: yeah

512 MARY: it's the list you don't know
513 it's this bit you don't know
      [indic. underscore characters]
514 this bit you're going to build

515 JULIE: but that's the same there isn't it?

516 MARY: why

517 JULIE: it's that bit you don't know
518 it's the second part you don't know
519 in the first bit isn't it
520 as well

The trouble they were having was referred back at this point to the underscore variable:

522 JULIE: no? Are we being confused by this underscore
524 do you think

526 MARY: yes

527 JULIE: I think we are aren't we?
528 'cos I'm sure we'd be able to do it if we had
529 ordinary variables there

Mary had another attempt at convincing Julie, and then offered her interpretation, and we see the extra variable Z appearing, to complete the mythical transitivity relationship (line 570):

566 MARY: it's going to go looking
through, through these assertions
for X and Y
actually it's more likely to have Z Y
[writing]
so that um it's got X all the time here

JULIE: yeah

MARY: it'll go up and down the list
first of all it'll come to Z
so it'll be X and Z

JULIE: yeah

MARY: and Z will get put in the list
then it'll go back up and it'll have X again
no it'll have Z at the beginning of the list
here as well as here

MARY: um I think all the time
it remembers what X is though
so it's looking for something that matches...

It was not clear what Mary was trying to get at, but she had, in
line 589, managed to weave into her 'theory' the matching process
she had identified earlier. Mary began to feel the weight against
her interpretation, and admitted to being confused. Julie suggested
that, according to this version of events, it was a recursive
definition after all:

JULIE: you're saying it is recursive then
is what you're saying

MARY: I think it is

JULIE: yeah

MARY: and it keeps snatchings things out of here
and keeps pushing them along
to get from X to Y

JULIE: presumably these two
these two things
go together don't they though
so how does this one react with this one then?

MARY: I think that that
if it is recursive

JULIE: Right, yeah
MARY: I think that is the stopping clause
so that when it's got X and X
it's not going to go any further
it's got what it wants

The final conclusion of the protocol was Mary's assertion of her interpretation:

MARY: it's finding a route from X to Y
it's finding the list
Together: between X and Y
if there is one

8.4 Comments

This study illustrates the determination with which students can stick to their abductive hypotheses. Mary was a shy, quietly spoken individual, who had done several protocols before. She was not known to the experimenter to be dogmatic or forceful in her opinions at all. And yet in this situation she defends her hypothesis vigorously, and gradually convinces herself that it is the correct explanation (cf Wason, 1962, discussed in Chapter 3). This is achieved in the face of opposition from Julie, and despite an explanation of the underscore character from the experimenter.

It appears that Mary had only a vague picture of a process in her head - the transitivity relation - and the difficult task was to get the code to support her idea. Whilst she was uncertain as to the function of the underscore, it was sufficient to indicate that 'something' happened, without being certain what. But even when she knew, this knowledge was not integrated into her interpretation, which appears to have remained intact and largely unscathed. Julie's approach of trying to fit variable instantiations with input and output is a very wise one, and it is this line of
questioning with which Mary is least able to cope.

Caution should be exercised to a certain degree in extrapolating too far from this protocol, however. Mary after all had put herself out on a limb by offering an explanation at all. In that sense she has exposed herself, and it may be that some of her determination is an artefact of social pressures to justify what she has said, or of her own quite understandable need to defend her opinion, or simply out of a sense of insecurity.

8.5 Discussion

We should bear in mind, however, that for whatever reason, learners frequently do create hypotheses abductively and theories are confirmed perhaps on the basis of a single piece of corroborative evidence (cf. Lewis and Mack, 1982). Mary was unable to accept the information coming from the experimenter because it did not fit with her story so far, and since it did not fit, it was rejected.

One of the problems for Prolog learners is that in the initial phases of learning, there is very little in the syntax which can inform them about function, control flow or the kinds of activities which are going on in the program. As yet, we do not know how best to teach Prolog, nor what a good virtual machine for the language looks like.

The only sorts of clues which can be informative are, for example, predicate names, which in this instance had been removed. Learners, then, are not only able to hallucinate meanings onto code, and abduct explanations of what they think it will do, in the way illustrated here, but in many cases they have to abduct such
explanations. If a learner has done this with some success, however, the teacher may find that when bugs show themselves eventually, they have very long roots, and correcting the error may require the unpicking of a complex web of abductions which the learner may want to defend. Simple presentation of correct information is, in this situation, unlikely to be effective, since learners may well ignore conflicting information.

This protocol study illustrates one abductive hypothesis which took place in response to a particular set of circumstances. Had Julie and Mary been able to run the code, their explanations would, of course, be very different. The task then would be to interpret the machine's behaviour and relate that to the syntax of the program, and in this enterprise, they would at least have had more information to work with. But the next chapter describes a longitudinal case study, and shows how prior knowledge combines with abductive reasoning, which has long term effects on students' perceptions of Prolog.
272

CHAPTER 9

A Case Study: Alex

9.1 Introduction

Lest it be thought that programmers experienced with other programming languages would escape from superbugs arising from superstrategies, we now describe the problems experienced by a student who participated in a case study. Alex represents quite a different type of subject than those so far discussed because he had learnt several other programming languages, but the study is important for several reasons. Firstly, it illustrates the role that prior knowledge plays in conceptualising a new domain, and the determination with which prior knowledge can be adhered to, even in the face of contradictory evidence. Secondly, many of the problems Alex experienced are the reverse side of what genuine novices go through - because Prolog is flexible, learners are not challenged in their interpretation, and misperceptions can persist for a very long time. Thirdly, the longer term effects that a fundamental misconception has on performance can be traced.

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 tries to construct a view of Prolog in which all operations are '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. To make matters worse, he was not being taught a detailed mechanistic view of Prolog (with which he might have been more comfortable) but was being given a high level introduction of
a broadly declarative nature, though it was not exclusively 'logical'.

Figure 9.1 illustrates the general strategy which Alex adopted, which was to interpret what was in fact a form of logical description of Prolog in terms of a mechanistic domain of the sort with which he was already familiar.

<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>REASONING</td>
</tr>
<tr>
<td>INTERP.</td>
<td>STRATEGIES</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>REFLECTIVE</td>
</tr>
<tr>
<td>GENERAL</td>
<td>Implicit</td>
</tr>
<tr>
<td></td>
<td>Practical</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
</tr>
<tr>
<td></td>
<td>Empirical/Behavioural</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td>Analytic</td>
</tr>
<tr>
<td></td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td>Hypothetical/Syntactic</td>
</tr>
<tr>
<td>MECHANISTIC</td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td>Analytic</td>
</tr>
<tr>
<td></td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td>Hypothetical/Empirical</td>
</tr>
</tbody>
</table>

Fig. 9.1 Alex's Pathway Showing Intuitive Moves from Logical Discourse to Mechanistic Discourse

The intuitive moves represented are either to interpret a logical expression as a mechanistic one, and/or to use reasoning strategies which are appropriate to the mechanistic level to reason about events which are in fact located within logical discourse. This strategy is not unreasonable, and, as is implied from the framework in Figure 9.1, having once made these intuitive moves, his explanations were subsequently plausible and 'made sense', but unfortunately they often did not contribute to his understanding of Prolog.

The bugs and superbugs that will be discussed are as follows:
* 'Abstract' Variable Bug

* 'Prolog is just like X' superstrategy
  * Data and Procedures Superbug
  * Conventional Variables superbug
* 'Lists are Brute Force' superbug
  * Unstructured Programs bug
  * 'Member' as a Procedure
  * Where is the Real Prolog?

Fig. 9.2 Bug and Superbugs for Alex

The superstrategy is 'Prolog is just like X' where X represents any previously learnt language. The major superbug is the data/procedures one, but this arises in the context of accompanying superbugs: one regarding the use of variables, and other regarding lists. These are described as superbugs because they were a direct importation of his prior experience with programming. The combination of the three superbugs - which constitute a sort of syndrome - produces misapprehensions about (a) program structuring, (b) how the 'procedure' member could possibly be conceived as 'predicate', and (c) a desire to get to grips with the 'real Prolog'. The Abstract Variables Bug, however, is regarded as a domain specific bug.

Because of the intricate dependencies between these bugs, after some introductory remarks, the domain specific bug will be discussed first, and then the two 'minor' superbugs (Conventional Variables and 'Lists are Brute Force'). This will then set the scene for the discussion of the major superbug of datas and procedures.

This study also illustrates the implausibility of one-shot learning (Anderson, 1982), where the student is supposed to be able to understand a class of problems after having analysed in detail a single case using a template. Alex went through several tasks
which, according to Anderson, should have played a part in
developing his understanding and eradicating misconceptions. But in
fact, they made no great inroads on Alex's interpretation of
Prolog, which was influenced by very deep concern about its
'sloppiness' and apparently undisciplined behaviour.

9.2. A Comment on Procedures

The methodology for the case study follows Anderson et al. (1982),
and was described in Chapter 5. Since this form of protocol
gathering produces enormous quantities of data, it will not all be
reproduced here, or in the appendices. Instead, the chapter is
organised around the major bugs which were identified in Alex's
protocol sessions. The data is presented as a series of snapshots.

Alex attended 11 sessions, covering the first 12 hours of learning.
During this time he covered the basic core of Prolog: facts and
rules; database accessing; pattern matching; backtracking;
recursion; list processing; and use of the cut. Example programs
to illustrate these features were a family tree program, a database
of artists and their works, a parts inventory for bicycles
(Clocksin and Mellish, 1981, p.54) and the development of this
program into a simple parser. Execution of programs was illustrated
through the use of a dynamic tracer which is part of the POPLOG
system, and the Byrd box models (Byrd, 1980) which provide an
account of control flow through clauses. Alex also did the first
Coombs and Stell (1985) backtracking exercise.

All of these topics were taught via the POPLOG TEACH files, except
9.3 Background Remarks

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, CFMI. 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).

150 BASIC and Fortran
151 and I worked for a while
152 with a database enquiry system that was based on cobol,
153 but all of these things,
154 all my programming languages I never
155 learnt them rigorously,
156 I only learnt them enough to get a feel for them,
158 so none of them are rigorous
159 and they're all sort of confused in my head.

171 .. in 1980 I was taught Fortran,
172 in 1982 I was taught Pascal.
173 Basic I more or less picked up on my own.
174 I ran a tutorial in Basic for psychology students
175 on an Apple microcomputer
176 and in the Cobol database enquiry system,
177 that was my last job just before I left New Zealand
178 with the city council
179 working with a thing called datatreive,
180 on a VAX, sorry it was on a PDPII/44.

He also recalled another assignment in a 'list-based' language:

188 One other thing I did do
189 was one and a half assignments
190 in a list based command programming language,
191 the Primos command programming language,
192 CFMI I think it might be called
193 is a list based language and I did an assignment
194 in that which I know is relevant to a lot of this stuff
195 because I did some fairly sophisticated stuff with lists.
196 I've just forgotten what it is.
197 That was a couple of years ago.

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 indicate 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.

A problem for him in the context of the case study was that his learning was not focused on a specific task. Or, rather, the task was to 'understand Prolog' in some abstract sense, rather than to employ it in a specific task. In later sessions, he exhibited some frustration with the theoretical aspects of Prolog, and was clearly unimpressed by it because he could not see how certain features could be useful to him.

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 unconvinced that learning Prolog was really worth the effort. The most frequently asked question, either implicit or explicit was: 'Why did they implement it that way?'. He seemed to feel that the tasks he was asked to do could have been done better in his other languages. Sadly, the experimenter failed to motivate him by providing a task which fitted into his conception of 'interesting'.

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 contribute to the program organisation problem, are 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 BASIC, 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.

9.4 Abstract Variables Bug

The first difficulty Alex experienced with Prolog variables was a common one, experienced by very many beginners, and was not directly a result of his previous experience. The problem was to establish a relation using three variables, and the predilection beginners have not to instantiate variables during clause construction can make the task very difficult, because abstract entities are being juggled.

He was writing clauses to capture the idea that one country could be 'contained' by another - e.g. Britain 'contains' Scotland, and Europe 'contains' Britain. Thus, Europe 'contains' Scotland. He was using several different kinds of clauses. The ground clauses (i.e. ones containing no variables) were of the form:

\[
\text{is in}(\text{scotland, britain}). \\
\text{is in}(\text{wales, britain}).
\]

The TEACH file showed how to build a simple rule defining a predicate 'contains':
contains(Country1, Country2) :-
   is_in(Country2, Country1).

Later, the learner is asked to define a new rule defining a predicate 'part_of' where 'part_of' can be used to capture the notion of 'nested' countries:

part_of(X, Y) :- is_in(X, Y).
part_of(X, Y) :- is_in(X, Z), is_in(Z, Y).

The situation is rather complicated now, since time has been spent creating the 'contains' predicate, which is not an essential component of the basic transitivity link - i.e. only the 'part_of' and 'is_in' predicates are needed. However, Alex, while constructing his rule, commented:

919 ahh to have a 'contains'...
920 Well the only example we've got of a 'contains'
921 is Scotland 'is in' Britain 'is in' Europe
922 so Europe contains Britain and Europe contains Scotland,
923 but that isn't clear from this one rule here, contains.
924 So you want to nest the contains rule
925 and something else which is the 'part_of' rule,
926 so 'part_of' is a more general 'contains',
927 which is possibly more than than one level deep,
928 so 'part_of' is just the more general 'contains' rule.
929 So I'm going to nest the 'contains' rule
930 within the 'part_of' rule,
931 and introduce a C3 as well as C1 and 2

This is quite a neat summing up - 'part_of' is more general because it may be more than one level deep; nesting 'contains' in 'part_of' ought to somehow provide the basic function for the more general relation. However, there is a problem in keeping the input arguments correctly ordered, as well as the added complication of the unnecessary predicate 'contains'.

But the way that Alex goes about it makes the task even harder because he uses uninstantiated variables which represent different countries, rather than giving them values whilst he figures out which order they should go in. One interpretation of the clauses
he wants to use are of the form:

Clause 1: is_in(<x>, <y>).

Clause 2: contains(<x>,<y>):- is_in(<y>,<x>).

Clause 3: part_of(<x>,<y>):- is_in(<x>,<z>),

is_in(<z>,<y>).

(where letters inside angle-brackets indicate some object)

These variable names are not optimal since they are abstract - i.e. there are no cues to help one keep track of relations. More helpful variable names can be substituted:

Clause 1: is_in(<small>, <large>).

Clause 2: contains(<large>,<small>):- is_in(<small>,<large>).

Clause 3: part_of(<small>,<huge>):- is_in(<small>,<large>),

is_in(<large>,<huge>).

This is rendered even easier by now making the prefix operator infix and expanding the operators:

Clause 1: small is_in large

Clause 2: large contains small if small is_in large

Clause 3: small is part of huge if small is in large and large is_in huge

Now the interpretation of the clauses in the program exploits our understanding of the 'real' words 'is_in' and 'contains' and 'part_of' to understand the ordering of size in the relationship. But in the attempt to ease the reasoning process, we have introduced a complication. Our interpretation fits the English interpretation of 'is in' - the first argument is the smaller object. In the 'contains' clause the first argument is the larger object - which fits our English language interpretation of the word
'contains'. But our third clause reverts to small object first, corresponding to 'part_of'. Looked at structurally:

Clause 1: predicate1 <small, large>.

Clause 2: predicate2 <large, small> :­
    predicate1 <small, large>.

Clause 3: predicate3 <small, huge> :­
    predicate1 <small, large>,
    predicate1 <large, huge>.

we can see that Predicate2 (corresponding to 'contains') somehow violates what would otherwise be a simple rule - small ones first, larger ones second. We can also see that Predicate3 ('part_of') does not build upon Predicate 2, which might imply some redundancy in the rule structures. But for the time being, Alex needed to bear this slightly anomalous status of the 'contains' predicate, otherwise he would have difficulties passing arguments around.

Alex said:

929 So I'm going to nest the 'contains' rule within the 'part_of' rule, and introduce a C3 as well as C1 and 2.

The facts given in the database were of the form:

is_in(small, large).

Prompted by the TEACH file, which referred to Country1, Country2 as variable names, Alex had a first attempt.

Clause 1: part_of(C1, C2):- is_in(C1, C2).

Clause 2: part_of(C3, C2):- contains(C2, C1),
    contains(C3, C2).

By substituting consistently in our naming scheme we get:

Clause 1: part_of <small, large> :-
    is_in <small, large>.

Clause 2: part_of <huge, large> :-
    contains <large, small>,
    contains <huge, large>.

This was intended to say that there are two ways that a small
country can be part of a larger one. The first (Clause 1) is simply when it is known that the smaller country is in the larger country. The second case (Clause 2) is when the small country is contained in a larger country and the larger country is contained in a 'huge' country. The first clause is fine, the second clause is not since it violates the argument ordering in its head. Part_of should follow the general rule 'smaller > larger', as in fact is correctly formulated in Clause 1. The calls to 'contains' in the body of Clause 2 are correct, in that they follow the convention 'larger > smaller'. The transitivity relation supposed to exist in Clause 2 is therefore corrupted: Huge is 'part_of' Large if Large contains Small, and Huge contains Large.

If we abandon the variable naming scheme, and read Clause 2 as an isolated specification, there appears to be a simple error in the head: part_of(Huge, Large) should read: part_of(Huge, Small). If we make this change, then logically speaking the clause would read correctly, with a convention that the larger country is always the first argument, and the smaller country is the second. This is turns out to be the reverse of our original naming scheme.

Alex's first attempt to fix the error was to change the argument order in the second clause to read

\[
\text{Clause 2: part_of(C1, C3):- contains(C1, C2),}
\]
\[
\quad \text{contains(C2, C3).}
\]

i.e.

\[
\text{part_of <small, huge> :- contains <small, large>,}
\]
\[
\quad \text{contains <large, huge>..
}\]

Again, this reads correctly from a logical perspective: Huge is 'part_of' Small if Large contains Small, and Huge contains Large -
but again, the naming scheme is violated since 'contains' in the program is defined to take the larger entity first. Rather than change argument orderings around again, Alex realised that using 'is_in' in the body of Clause 2 would avoid further problems.

This kind of problem can hold beginners up for a very long time. In this particular case, Alex was led astray by the rather confusing instructions in the TEACH file. But it is difficult to mentally simulate complex relations, and therefore it can be hard to identify which clause, or variable name is the culprit when clauses such as these do not work as intended. This seems to be simply a mechanistic bug, rather than a superbug: the task is hard, and it is easy to get confused. As has been established in the literature on deductive reasoning, abstract expressions do make reasoning difficult. The task can be made easier by using helpful variable names, but this tactic must also be adopted with caution, since it can encourage students to lean too heavily on natural language meanings.

9.5 Superstrategies and Superbugs

As was pointed out in the introduction to this chapter, the main superstrategy Alex used was to try to understand Prolog in terms of languages with which he was already familiar. Since none of these languages belonged to the same 'family' as Prolog, his overall success was limited. On the other hand, Alex was able to construct plausible explanations for some parts of processes he observed, and although he was sometimes incorrect in his first approximation of what was going on, he was at least heading in the right overall direction. The particular superbugs which arose from the strategy
are described below.

**Conventional Variables Superbug**

This is another difficulty with variables, but this time the problem was directly related to Alex's previous experience of other programming languages.

The following extract is taken from a session in which he had to construct a program as specified in Figure 9.3.

Michelangelo is a painter and sculptor. Rafael is a painter. Leonardo is a painter and a sculptor. Bach, Handel and Haydn are composers. Leonardo painted the 'Mona Lisa' and worked on the Sistine Chapel. Michelangelo sculpted the 'David' and painted the ceiling of the Sistine Chapel. Rafael also worked on the Sistine Chapel, and painted the 'Madonna and Child'. Bach composed the 'Toccata'. Handel composed the 'Water Music' and Haydn composed the 'Creation'.

The program should be able to tell us what the profession of any given individual is, what the name of their works are and whether each work is a sculpture, a painting or a piece of music. We should also be able to ask what works the program knows about.

**Fig. 9.3 The Artists Program Specification**

Having constructed some clauses which stated:

- Clause 4. isa(painting, X) :- painted(Who, X).
- Clause 5. isa(statue, X) :- sculpted(Who, X).
- Clause 6. isa(piece_music, X) :- composed(Who, X).

Alex was about to type:

\[
\text{works(Who, What):- made(Who, What).}
\]

when he suddenly stopped and commented:

234 Will I get into trouble -
235 I will get into trouble
236 I'll get into bags of trouble -
237 because I have two variables 'who' 'what' floating round.
Because the scope of variables is local to clauses, this in fact does not matter. They will each be treated as individual items. However, of course, in other programming languages variables are often globally scoped. Alex decided, therefore, to create the clause:

```prolog
works(Who2, What2) :- made(Who2, What2).
```

However, he did not seem concerned that he was using the variables 'Who' and 'What' in the 'made' clauses above. The variables in the predicate 'made' were sensibly organised, but all they achieved was the creation of a new predicate 'made' which perhaps was supposed (optimistically) to generalise over 'painted/sculpted/composed'. The 'isa' predicates in Clauses 4 and 5 use X as the variable name in the head and body (indicating some appreciation of pattern matching), but also use the variable name 'Who'. It may be that he thought the body of 'isa' (corresponding as it does to the body of 'made') can afford to use the variable name 'Who' since in the 'isa' rule he was chasing another piece of information - the name of the work. So 'X' in that instance had some sort of special status since it did not occur elsewhere in the program.

Having typed in his 'works' rule, Alex commented:

239 You probably don't even need two variables in there.
240 I presume you can have a rule with only one argument,
241 say, made(X) if anybody....
242 or works(X) if anybody did that.
243 Well I'll do it this way first.

which was reasonable, but also indicated that he had forgotten that Prolog needs the extra variable in the head to match with the instantiation in the body, and produce the output. He was more concerned, though, to 'suppress' extra information which was provided in his output, which involved re-writing the 'works'
clause as follows:

\[
\text{works}(Q) \leftarrow \text{made}(\text{Who2}, \text{What2}).
\]

which the produced the output 'Q = _1' for the simple reason that
the variable 'Q' is 'floating'. It does not match with anything,
and so obtains no instantiation. Given that in previous clauses,
the relationship between variables in the head and those in the
body had been correctly sustained, this would seem to constitute a
'slip'. Alex thought it may be an error, and whilst the
experimenter was talking about the nature of underscore variables
in Prolog, Alex realised what was happening:

340 Negative 1 might flag an empty variable,
341 or unassigned or something
which the experimenter missed. She suggested going back to the
previous formulation (i.e. putting another variable back in) to
compare the output. Alex inserted a further variable (W) which
acted in exactly the same way as the first, producing output: Q =
_1, W = _2. After examining the clauses again, Alex realised what
needed to be done:

398 I know what I want to do
399 I want to put What 2 in there (indicating W).
400 That's what I wanna do.

This remark occurred without discussion. The new clause, therefore,
was:

\[
\text{works}(\text{What2}) \leftarrow \text{made}(\text{Who2}, \text{What2}).
\]

Having resolved the problem in that instance, the concern arose
again later. He began to type in his query:

but stopped with the remark:

245 I have used those before
246 so I might get confused further down the tree
[referring to X variable in the query]

247 Let's use something completely different
248 just to be on the safe side.

[types ?- works(Z, A).]

Here the concern seemed to be that he (rather than Prolog) would not keep track of variable names if he used the same ones. This shift of concern might indicate that he viewed the naming of variables for the purposes of output as very different from variables used by the program. This distinction would naturally spring from his experience with other languages.

His final question was:

457 This would have died in a screaming heap
458 if I had just got What there instead of What2.

The experimenter explained that it would have been fine because variables scope across clauses only:

468 Eughooo!
470 I don't like that!

was the response.

The interesting facet of this protocol fragment is that there appeared to be some inconsistencies in the way he viewed variables. Sometimes he seemed to be able to use them as Prolog variables, and indeed, he eventually sorted out his own difficulty here. But at other points, he suddenly seemed to recollect something from his previous experience, which was applied to the particular variable he was considering at that point. The close relationship between superstrategy and superbug can be seen in episodes such as that described above, where the application of prior knowledge is not necessarily as useful as one might imagine it would be.
'Lists are brute force' Superbug

This protocol fragment illustrates the determination with which Alex stuck to his ideas about what was and was not good practice, and also provides a clue to another bug which is discussed later. Recall that Alex had told the experimenter that he had done some 'fairly sophisticated stuff with lists'. The scene was set for misinterpretation, because the experimenter thought he meant lists in the Prolog sense, which he did not. Therefore, when the experimenter suggested (in the same exercise as above) that lists might be a useful way to represent the various works of a given artist, Alex, somewhat to the experimenter's surprise, replied:

135 But that's sort of bloody-minded.

EXP: Why?

137 Well it's a brute force solution.
138 There should be a way of collecting painted, composed and ....
140 painted and sculpted are the relevant ones.
141 There should be way of collecting painted and sculpted into one category ....

At this point, Alex clearly had the solution, but refused to use lists which the experimenter found odd. She commented, in the hope that he would want to find out about lists:

EXP: At the moment you're working with just facts, so anything you do at the moment is going to be brute force....

but his response was:

150 Right. I want some rules.

Alex started typing 'made(X..)' but deleted it, clearly unable to conceptualise how to collect multiple items of data together. The experimenter persisted with an explanation of the basic syntax of
lists, and the way in which information can be grouped. That Alex's conception of lists was different from Prolog lists was hinted at after the syntax had been described:

163 Right.
164 Ah can you give the list a name, or....?
165 I mean how do you use a list?

Despite the explanation, he responded:

182 There should be an easier way to do it
183 just with rules,
184 but I've forgotten
185 all about rules..

193 There should be another
194 way of doing it without lists,
195 but I'll use lists if I can't think of another way.

This is an interesting episode because Alex abandoned his role as 'recipient' of information and takes control. He rejected the experimenter's less than subtle advice to use lists, insisting that he could do it some other way which would be better, although he was aware that Prolog did not behave in the same way as any other language he knew.

After struggling with the exercise for a while, he commented:

429 These isa's didn't really capture the generalisation
430 I was being asked for did they?

thereby in some senses acknowledging his own difficulties.

Note in the protocol fragments the first evidence of the major conceptual bug which is discussed next: the data/procedures distinction. After the description of lists Alex asked:

163 Right.
164 Ah can you give the list a name, or....?
165 I mean how do you use a list?

indicating that he was expecting to give lists a name, and to write
procedures which would act upon them.

Data and Procedures Superbug

The bugs described so far presented Alex with various degrees of difficulty, but to a large extent, they were gradually dealt with. However, this bug was persistent, and permeated even the very last exercise Alex had to do. It stems from the idea that data is static and procedures are active, which is a conviction left over from Alex's 'scientific programming'.

The first signs of the data/procedures issue arose in the first session. This constituted Alex's first brush with Prolog, and comparisons with other languages were very much to the fore. Initially it worked in his favour in relation to certain basics, particularly in relation to surface syntax issues. For example, in the first 100 lines of the protocol, he had identified: that facts end with full stops, that variable names are capitalised and can be of any length, and that spaces are not important in Prolog expressions. He also has no difficulty understanding by what route Prolog goes through the database, deduced from output.

But the higher levels of abstraction were problematic, as he acknowledged in his opening remarks.

1 Its just the stuff about non-reciprocity of relationships.
2 I make too many assumptions
3 from programming languages from
4 from Pascal
5 which is the one I know best.
6 Like it horrifies me
7 that you can have all sorts of data types in lists
8 and declare variables in the middles of programs...

Note that Alex was aware of his tendency to 'make assumptions' - he pointed this fact out at regular intervals during the study. He
also commented in this first session that he liked to 'jump in the deep end'. Clearly, he had learnt from scientific programming a methodical way of organising data, datatypes, procedures and using variables. This background also governed what he considered to be a good feature: for example, he was slightly taken aback that Prolog had no keywords.

But Alex's ability to understand at the lower levels may have actually created a trap for him in the sense that he assumed he had understood things because there was little to confront him, or engage him, at those lower levels. Prolog syntax seemed more or less straightforward, and he was not alerted in any way to the significant differences between the concepts he knew and the concepts he was being confronted with. This is a similar problem to the one experienced by students using natural language to interpret programs, except the source and its effect are rather different. In the natural language case, the learner can 'get away' with construing meaning from clauses; in this case, Alex thought them very simple, and did not therefore find it necessary to pay much heed to what he was being taught - it all looked easy.

Alex worked through the TEACH file rapidly, then with only a few queries, and said that, those queries aside, he had no problems.

However, when the experimenter, in summarising what had been covered so far, referred to 'the program' he had written, he responded:

202 Well I mean I wouldn't really call it a program,
203 I'd call it a datafile,
204 but that's just my bias.
205 I'll believe it.
The blurred distinction between data and procedures in Prolog is a tricky one, particularly for programmers in other languages. To them it appears undisciplined, 'half-built', messy, and therefore undesirable. They have been trained out of mixing them up, viz. a later comment:

464 We get our hands smacked in Pascal for jumbling up our data and procedures

So in one respect, Alex's reaction was to be expected. What is remarkable, though, is the dogged persistence with which he stuck to his own interpretation despite flagging attempts on the part of the experimenter to explain the difference. She made the point that although the program might resemble a datafile at that moment, in time, it would not only contain simple 'facts' - information would be implicit in the program which would need to be deduced. After a lengthy explanation, Alex decided that it was a 'slightly different interpretation' on familiar words. For him the issue was simply resolved. That it was not resolved was evidenced a little later in the session when the issue was raised again. He asked for the 'formal definition' of a rule, a question which threw the experimenter a little, and, in her reply she mentioned recursive rules. Alex responded immediately:

416 But that's a rule in a sort of...
417 in an active sense
418 its not a rule within the database

and after further explanation:

429 .. so you can write recursive rules
430 for Prolog databases
431 or for Prolog programs?

This slight confusion may have arisen from a previous comment by the experimenter about the relationship between Prolog programs and
databases. But it also illustrates how Alex pursued a line of enquiry doggedly, and is reminiscent of Mary's adherence to her interpretation of the 'member' clauses. However, the source of the problem emerged eventually:

446 The sort of distinction
447 that I think I'm confused about,
448 like I always make a distinction
449 between sort of data information,
450 and procedures/processes,
451 and for me recursive is something
452 procedures/processes do
454 and not something data/information does

Adopting this view can contribute to a difficulty that other Prolog novices have which is that it can be difficult to understand how anything 'gets done' in Prolog programs. If data is 'inactive', then clauses which function mainly by matching alone may be virtually incomprehensible. This is also a symptom of not yet understanding the underlying mechanisms well enough to interpret Prolog's behaviour. This is particularly difficult if the declarative semantics are being used.

However, it is obvious why it would cause Alex problems at this point: his understanding of programs was that they consist of procedures which work upon data. The experimenter had already tried to explain that Prolog structures act upon and with other structures, but did not succeed in convincing Alex of the value of this enterprise.

But the major point is the effect that this view had on Alex's subsequent work. Because of the division that has been forced between data and procedures, he found it very difficult to understand how best to structure programs. He tended to put a great deal of information in explicitly, and because of his problems with
lists, he never really managed to get a good structure into programs. This difficulty persisted right through until the end of his case study. Some examples of his approach are provided below.

A week later, Alex was again trying to construct the Artists program. His first comment was:

15 I tend badly to do things from the bottom up –
16 like I'll put a lot of things in
17 which I may change later.

In fact, what happened was that he began by entering the basic facts, and did not go back and restructure. If we look at the final program we can see the lack of proper structuring.
isa(mich, painter).
isa(mich, sculptor).
isa(rafael, painter).
isa(leon, painter).
isa(leon, sculptor).
isa(bach, composer).
isa(handel, composer).
isa(haydn, composer).
isa(artist, painter).
isa(artist, sculptor).
painted(leon, monal).
painted(leon, sistinec).
sculpted(mich, david).
painted(rafael, madonna).
painted(rafael, sistinec).
composed(bach, toccata).
composed(handel, watermusic).
composed(haydn, creation).


isa(painting, X) :- painted(Who, X).
isa(statue, X) :- sculpted(Who, X).
isa(piecem, X) :- composed(Who, X).

works(What2) :- made(Who2, What2).

Fig. 9.4 Alex's Artists Program

First, there is rather a lot of redundancy in the program not only in the number of explicit facts it contains, but also in the rules. The 'made' predicate and the 'isa' predicate all access the same facts. Alex noted this point, and thought that he should be able to group these rules up somehow, since both clauses refer to the same facts in the bodies. But instead of thinking how to conflate the rules, perhaps by the use of conjunctions in the body for example, or lists, he began to introduce yet another rule:

works(What2) :- made(Who2, What2).

Due to his reluctance to use lists, it was difficult to see how he could compress information. Although most solutions were found, it was only by forced backtracking. He commented:
We do have the problem of the recurring thingumyjigs. It would be horrible to sort out and I would rather do it in the form of the question rather than the form of the database. I would rather write a bit of code for the question which says print it out if you haven't printed it out already.

In the light of the remark in line 414, it is clear that the question of structure had not been foremost in Alex's mind. After a brief discussion about the uses of lists, Alex conceded:

A month later, Alex was working on the the program for a parts inventory for a bicycle (Clocksin and Mellish 1981). The program builds assemblies from basic parts by creating lists of subparts. An assembly may consist of several sub-assemblies each of which need to be constructed from parts (see Appendix 6 for a transcript of the program, and for copies of Alex's programs). Whilst reading through the program Alex commented:

Why are you inventing a predicate, why not just use assembly with nut if and only if basicpart nut. Why invent this new predicate partsof?

I would have used constituents - its just a special class of assemblage.

In other words, from the very outset, he was wanting to reduce the amount of structure in the program by eliminating the predicate 'partsof'. The experimenter pointed out that it might be a useful intermediary structure, but he commented:

It just doesn't seem to be the way I would have done it, but never mind, they've invented partsof so...

The structure of the program reflects the typical goal/subgoal
recursive approach of Prolog, and it seemed that Alex was still not yet into that way of thinking. This becomes particularly apparent in the follow-up exercise, which was to adapt the inventory program to be used as a small parser, as recommended in the book.

Instead of gradually replacing basic parts and sub-assemblies with linguistic structures, Alex, after some fiddling about, decided:

579 ...I think we might try
580 not to take the analogy with that program too seriously
581 and just do it again....

and later, having obliterated most of his original inventory program:

774 I wish I hadn't eliminated so much of that now......
775 I can't remember whether I actually made a mistake
776 in labelling those at all.
777 I don't think I have.
778 it seems to me I've got two different sorts
779 of assemblies now,
780 assemblies that make reference to basic parts
781 and assemblies which make reference to other assemblies
782 but that's what assemblies did before didn't they

The final program that he produced (see Appendix 5) was very limited, and most solutions needed to be obtained by forced backtracking. Alex commented:

893 ... it didn't impose any structure on that at all,
894 and there's no termination
895 [continues forcing backtracking]

The session ended abortively.

It can be seen from the above protocols that Alex is not simply suffering from a lack of knowledge. All the necessary tools needed to construct programs had been made available to him, either in TEACH files, from the text book, or by the experimenter. But his interpretation of the various tasks was heavily influenced by his
previous computing background. This emphasises the point made by Lewis and Mack (1982, op. cit.) that novices construct the problem domain for themselves, and to some extent select out what they think they need, rather than what they do in fact need.

'Member' as a Procedure

This section illustrates a consequence of some of these superbugs: it is partly associated with the the data/procedures bug, and partly associated with his problems with lists. Alex found it hard to understand how an operation like membership could possibly be defined as a predicate:

1 How do you use a thing like member as a predicate?

Membership perhaps appeared to him to be a procedural concept, because of its involving recursion. Alex had no trouble conceptualising the process:

303 ...to check if an element is in a list
304 they want you to perform a recursive function on the list
305 checking the head for identity until you reach the empty list.

although his vocabulary gave him away - a recursive 'function'. He was also unimpressed at having to define the clauses:

307 They're going to make us construct the member
308 routine from first principals?
309 Oh dear.

The lists bug emerged due to the experimenter having provided an example earlier of a list preceded by a predicate. The predicate was interpreted by Alex as the name of the list. The experimenter explained that the 'name' was a simply a predicate, at which Alex commented:
Ah that's why I was confused because I was conceptualising foo in front of the list as giving it a name.

I'll have to have a think about that, about precisely what kind of a thing a list is. So basically a list is just a.....

EXP: Datastructure...

something you can put in a single item, in a fact, or a clause, a fact or a rule which is the substitute. So you have got to build the structure of the fact or the rule around it.

This seemed to indicate that he was beginning to understand lists, but when Alex had entered the member clauses, he typed this query:

\texttt{oof(a,b,c,d,e,f)}.

When asked what he expected to happen he replied:

\begin{itemize}
\item[434] Well as far as I understand it at the moment I've got a dummy predicate on a one argument rule and the rule consists of just one argument which is a list abedef, in much the same way as up there we had a rule consisting of two arguments both of which were lists.
\end{itemize}

EXP: Right. So this is your query.

\begin{itemize}
\item[441] No that's not my query. That's just entering the datastructure for me to make a query about.
\end{itemize}

\begin{itemize}
\item[445] So now what I would expect to be able to do is to make a query member\,(a,oof).
\end{itemize}

He was again making a distinction between data and process - he needed to enter the data at a different time in order that later he can ask questions of it. The experimenter tried to explain:

\begin{itemize}
\item[462] Ah. That seems almost trivial
\end{itemize}

EXP: ... my plan was for you to ask, what if it's not the head of the list.
And we'd go on. But the problem is we've already had the lecture on building member in POP so I more or less know what to expect, so I probably wouldn't have asked that question.

Again, Alex was mistaking his understanding of the process of establishing membership in a general sense for understanding how Prolog goes about it. But the crucial problem that Alex still seemed to be ignoring was that Prolog has underlying mechanisms which makes its execution very different from other languages. Finally, Alex commented on his misperception about list naming:

I think I'm with you. The way I was visualising it was that that [foo] was defining a rule, or defining a function, and once we have some data we can call the function on the data.

Interestingly, Alex arrived the next day saying:

I think I realised why I was having problems with this thing. I was trying to treat member as a function which calculates something, whereas in fact its just a predicate which matches something isn't it?

Just like any other predicate, and I could call it whatever I liked.

It's not a system inbuilt function.

And instead of the machine having a library of routines and functions to call on, its simply got a basic ahm database - a thing with a lot of predicates already in it. And I've got to think of these predicates in terms of pattern matches, not as functions completing things...
That's where my problem was, I was trying to think of member as a function to which I passed routines - or passed data and it did the matching and popped them back. But it's just like any other predicate.

Well, I've got rid of another set of preconceptions that came with me from my scientific programming languages.

This problem, then, appeared to be eventually overcome at some level, but unfortunately the opportunity to discover whether or not he could use 'member' correctly was lost by the experimenter.

It has been shown how the difficulties with data and procedures persistently cropped up in Alex's work, even in his last session. He did recognise that things needed organising better, but this acknowledgement did not seem sufficient to open him up to ways of structuring programs in Prolog. The mixtures of insight on the one hand (i.e. the 'member' episode) and sticking to old habits on the other was frustrating.

Where is the Real Prolog?

Alex fundamentally wanted to know the 'truth' about Prolog, which it is not possible to provide straight off for beginners. He seemed to be unhappy at working at a conceptual, virtual level, but was very much taken with the tracer when it was introduced. It became clear at one point though, that he thought this was the 'real' Prolog.

Alex was watching the dynamic tracer work through the append clauses. The dynamic trace package used is one which can only be used effectively on small programs. It responds by drawing a tree illustrating each step of the execution (see Figure 9.5).
append([a,b],[c,d], [a|_2])

append([b],[c,d],[_2])

The tracer waits until a key is pressed before it takes the next step which allows tutors and students to establish exactly what has just happened before proceeding to the next event. The information provided by the tracer is very comprehensive. The names of predicates and the number of the clause being used appears in the tree, along with both instantiated and uninstantiated variables. Variable bindings can be seen to happen dynamically, and the command line gives information as to whether clauses are being called, retried or failing.

append([a,b],[c,d],[a|b|[c,d]])

append([b],[c,d],[b|[c,d]])

append([], [c,d],[c,d])

Alex became anxious about how the way that 'Prolog' (as represented in the tracer) recursively moved down clauses:
We tell it to pick out X and to put it there

EXP: Which is what it does

But why doesn't it leave it where it was also?

Ah - surely...

And it says move, why doesn't it just say copy?

why does it say delete the old X?

EXP: Well because you're asking it - what do you mean?

Well I would have expected it to copy 'a' up there

but I would also have expected it to leave 'a' down there,

I mean when you bind variables

you don't remove them from their homes

In a later session, however, he realised that the tracer was not a window onto 'real' Prolog, but a representation:

So we aren't really seeing a window into Prolog here,

we're seeing a distorted window into Prolog,

or a dirty window into Prolog

Similarly, he was concerned about the representation of Prolog in Byrd box models (Byrd, 1980). The box models are a way of describing how Prolog proceeds through some computation, and how certain types of clauses and operators affect the flow of execution. However, Alex seemed to do the same thing with this representation of Prolog as he did with the tracer -

OK now I have a real problem about this

because I would have thought that your machine

comes chugging along and it hits the top line,

and it sets up a box for alpha and beta

and then it goes chugging through there and sets up a box with gamma and delta in it

since it's always seemed to me before

that we test one line and

then we test the next line

but you're saying that its smart enough

to look through and read all the lines

that have got test on it

and set up one big box for all of them.

Alex was quite justified in his concern, of course, because he was
right. Prolog was not creating boxes and putting things in them. Box models are a representational schema. However, these interludes suggest that we must be very careful about the models we use with beginners to describe Prolog and its functions, since features of the representation may be taken up as part and parcel of Prolog itself.

9.6 Discussion

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 'data and procedures' misconception seemed to be at the root of other difficulties, and it was the one that Alex found the hardest to shake off. It was, however, compounded by interaction effects from the variable bugs and misconceptions about lists. Apart from Alex's idea that lists were 'brute force', a misapprehension which arose from his definition of the term 'list', the idea of using
lists as high-level data structures 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.

9.7 Conclusions

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

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

In this thesis, observational studies of novices learning to program in Prolog have been reported and an account of some powerful, general purpose strategies which novices bring to bear on the programming task has been developed. These have been dubbed 'superstrategies' which are essential for the beginner to make progress in an unfamiliar domain, but which, equally, upon occasion, may lead to superbugs (Pea, 1986). The jumping off point for superstrategies is existing knowledge - often tacit knowledge - which the novice uses to interpret incoming information. Abductive reasoning (Lewis and Mack, 1982) is used by beginners to knit prior knowledge together with existing 'knowledge' (or beliefs).

A theoretical framework based on a three-levelled discourse model has been described and elaborated, which can form the basis of further work in this area. The discourse framework shows that the potential for superbugs in programming is quite large, but two superstrategies in particular have been investigated through observational studies in this thesis. The first is when students use natural language semantics to understand Prolog. This superstrategy is linked with the declarative interpretation of programs. The second superstrategy is when students conduct their dialogue with the computer as if the machine were another person with human inferential capacities and insight.

The studies illustrate effective applications of superstrategies,
but they also show how superbugs can arise. Superstrategy and superbug analysis is thus shown to be useful because it sets the task of learning to program in a broader context through juxtaposition of both failures and successes derived from the same basic strategy. This in turn points towards a more balanced understanding not only of the sorts of difficulties experienced by beginners undertaking this complex task, but also of the ways in which people are equipped to deal with such complexity.

In this chapter some general comments about the above points are made, and their implications for teaching Prolog is discussed. Subsequently, suggestions for further work are presented.

10.2 General Comments on Intuitive Strategies in Formal Domains

The empirical studies reported help underline the points made in Chapter 2 where it was suggested that learners are not empty vessels contributing little to the learning situation. Learners come armed with general purpose strategies to interpret new information in the light of existing knowledge, whether or not that knowledge is relevant. One very powerful set of strategies has been examined in detail, those typically associated with human discourse interpretation. Such strategies lead to certain kinds of assumptions which are taken for granted, both in terms of the structure of the discourse and in terms of its language. It has been shown that formal discourses (such as programming) violate many of these tacit assumptions, which at best means that students labour under misapprehensions, and at worst means that students become confused.

However, the most important point is that superstrategies are often
robust and effective means by which the students observed in this study interpreted the novel domain of Prolog programming and which at least allowed them to do something sensible even if it was not precisely correct.

Empirical studies of three types of learner have been presented. One group had been taught a broadly declarative interpretation (i.e. one in which computational mechanisms were not emphasised); another had been taught a procedural interpretation of Prolog. A case study has also been presented which focused on a student who had learnt other programming languages to see to what extent he had superbug difficulties. An illustration of abductive reasoning was also presented.

The context for superstrategy use is one in which, as they tried to make sense of the various tasks with which they were presented, students drew on whatever knowledge they could muster to to construct a kind of 'story' or rationale for their reasoning. Although the stories drew from a variety of knowledge sources - where correct information provided within course work was only one source - stories were not randomly strung together. For example, an explanation of a program's execution might draw upon knowledge derived as follows:

* imagining what a person would do next
* integrating any known fragments of information (e.g. Known fact: 'Prolog is exhaustive in its execution strategy' - knowing this, do whatever one could possibly construe as 'the most exhaustive' thing)
* gleaning information from variable names and clause names to help with the explanation
* incorporating into the explanation any other idiosyncratic beliefs held by the individual.
It is interesting to note that students - particularly in the MSc group - sometimes seemed to cling to slogans which were brought from lectures or textbooks. Typical slogans for Prolog are things such as: 'NEVER backtrack over the cut', or 'when backtracking ALWAYS go back to the last choicepoint'. Although these slogans might be re-iterated by students during a session they were not always correctly interpreted or applied. Fragments of information such as these need to be fitted together coherently to form a 'good' story, where 'goodness' is determined by the story-teller and may or may not accord with actual facts. The stories constitute, then, personal accounts or models of the process to be understood by the student.

This sort of behaviour seems consistent with di Sessa's (1982) notion of 'distributed encoding' (cf Chapter 2) - recall that this was a functional view of knowledge (as opposed to a formal view) which involves 'a confluence and complex orchestration of a large number of partial understandings, with many of them based on previous knowledge' (di Sessa, 1982, p. 59). Distributed encoding is characterised as fragmentary, and it often seemed in protocol sessions in the studies reported here that students were gathering what they had in the way of partial understandings in an attempt to pull together a coherent whole. In this respect, interpretations were newly constructed during the session. This implies that an interpretation generated on another occasion may contain similar elements, but differ in details, from the current 'story'.

It should be noted that some of the exercises set in protocol sessions were novel for all the students who participated - e.g. they never had thought about Prolog execution in such detail as was
required by the Coombs and Stell backtracking exercise. Clearly, then, these students needed to work out what would happen as they went along. But the same sort of behaviour was also evident in the Computers and Thought group when, after having performed the relatively familiar exercise of clause construction, they were asked to explain what they themselves had written. Novelty alone, therefore, does not account for 'story-telling' behaviour.

These observations are consonant with the genetic account of learning in adults, described in Chapter 2, in which there are necessary phases of understanding where the role of personal world models (or stories) is crucial to the eventual development of 'real' understanding. It was stressed that identifying such 'genetically antecedent partial understandings', which can later be shaped by the teacher, is a crucial pre-requisite to understanding how best to teach formal subjects. In the circumstances observed in protocol sessions students did seem to use natural language and human discourse interpretation strategies as knowledge frames for interpreting and performing the given task in a similar way that di Sessa claims English can act as a knowledge frame for understanding LOGO syntax (cf Chapter 2). Nevertheless, his caveat on this topic is worth repeating to highlight the superstrategy/supert>ug paradox:

No learner believes LOGO is English for very long. But we must be aware that globally destructive misconceptions may be fostered as well as misconceptions which, ironically, are profitable. (p.13)

It has also been emphasised in the thesis that simple presentation of correct information does not seem to be sufficient to guarantee the genesis of correct models of Prolog execution in students, mainly because such information must be interpreted. It is during
the process of interpretation that extraneous information may be introduced, whilst necessary information may unwittingly be cast aside.

In other words, although the refinement of teaching models is always useful (if not essential) it should be not be assumed that such efforts will prevent students from misconstruing the to-be-learnt material. Students in our studies were often known to have had access to correct information not only from text books, but also from lectures, TEACH files and tutorial interactions which had been monitored and attended by the observer. Nevertheless, this information frequently did not find its way 'uncontaminated' into the explanations offered by students, who seemed more concerned with the telling of a convincing story, improvising where necessary to maintain continuity. Rather than viewing this as a flaw in novice performance which should be eradicated, it seems preferable to regard it as a sensible learning strategy, in which pieces of knowledge are given coherence by their context within a story which has been created by the learner herself, and indicates the active engagement of an individual with a body of knowledge, rather than passive ingestion of information.

The construction of a personal theory of a novel domain or novel task has a respectable history in psychology and Artificial Intelligence. People do not accumulate disparate pieces of information, but tend to organise their knowledge into often quite coherent, but naive, 'theories' (di Sessa, 1977). The development of the individual's understanding of formal knowledge has been described in similar terms to those applied to the development of scientific concepts in scientific communities (e.g. Ruhn, 1962,
Popper, 1976). Just as scientific theories have a natural progression, so there are essential 'developmental' steps in personal 'theory evolution'.

From this view of learning, then, the role of misinterpretation (and therefore of 'bugs' or errors) is changed. Rather than focusing simply on the statistical frequency of errors and/or resulting buggy programs, the broader perspective gives some insight into underlying causes of certain types of error. At the same time it is acknowledged that some misinterpretations are, in fact, healthy, and could not (or should not) be eradicated on the grounds that to do so may well be more confusing than helpful for the novice. Developmental, or genetic, views of learning need to take into account the learner's present state of knowledge (i.e. trying to accelerate individuals beyond their developmental capabilities can be counter-productive).

Furthermore, in this view, subjects can be provided with the opportunity to comment on their own work because the observer needs to know why the subject thinks as she does. This can yield valuable information with regard to the status of apparent misconceptions, some of which can be strikingly plausible once the line of reasoning which led to them has been explained by the student to the observer. Also, in these studies, students themselves often confessed that they were unhappy with their own 'story' generated during the protocol session, and were quite well able to pinpoint the inadequacies in their account. This led to student-directed questioning of the observer, which in turn led to improved understanding. This implies that in these cases the students were their own best teachers provided that the task convinced them that
there was something there which they had not thought about in detail before.

Superstrategy 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 by insidiously contravening the sorts of assumptions which it encourages students to make (e.g. a programming language which looks like English but which bears no relation to the English language in its semantics).

10.3 The Discourse Framework

The discourse framework has been used to structure the discussion of superstrategies and superbugs. The main principle underpinning the discourse framework is that each of the domains which bear on Prolog is 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 potential difficulty for novices is that they do not understand what the implications of moving from one discourse to another are, and what constraints govern the methods available for problem representation or problem solution. Due to this lack of knowledge, intuitive strategies are used, which affect the novice's perception of the problem to be solved, and appropriate ways of solving it.

The literature pertaining to formal reasoning in the logical domain was used as a basis for developing this particular view of learning to program. This literature shows that, for untutored people,
formal logical reasoning is typically error-prone because the non-
logician uses real-world practical knowledge and ordinary
comprehension processes to interpret and solve problems. The
'meaning' of problems, therefore, is inappropriately derived from
natural language semantics. We have suggested that there is an
analogous situation in programming, where students similarly import
into the domain a great deal of intuition and real-world knowledge.

The basic framework, shown in Figure 10.1, illustrates what kinds

<table>
<thead>
<tr>
<th>Name of Discourse</th>
<th>Components of discourse</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT V INTERP.</td>
<td>REASONING STRATEGIES</td>
</tr>
<tr>
<td>GENERAL</td>
<td>Implicit</td>
</tr>
<tr>
<td></td>
<td>(Natural)</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td>(Logical)</td>
</tr>
<tr>
<td>MECHANISTIC</td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td>(Programming)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 10.1 The Three-Levelled Discourse Framework

of interpretation are appropriate at different levels and within
different phases of the problem solving process. Most of these
terms (e.g. analytic/practical, empirical/hypothetical) have been
derived from the deductive reasoning literature, although the terms
'implicit/explicit' have been used in this study to distinguish
between formal and natural language interpretation.

This view of the programming task has provided a useful structure
for both the empirical observations and subsequent discussion. The
most important facet of the framework is the fact that it represents the three discourse levels, and these remain crucial to the arguments presented in the thesis. However, the division of problem solving in each discourse into four components has not been fully exploited in the present work. It seems intuitively plausible that the choice of strategy at different phases during problem solving/program writing will vary - e.g. the use of natural language semantics during empirical program evaluation may be less than during program creation - but this has yet to be fully demonstrated.

The analysis of performance according to the framework has been conducted by noting the type of language that a given subject has used to describe a particular event. It has been argued that each discourse has a particular focus, and that a 'discourse appropriate' description or explanation does not use concepts which belong in another domain. For example, the word 'list' at the general level of discourse could have a variety of meanings depending on context, but a reasonable candidate meaning might be as in 'shopping list', indicating a collection of items to be obtained. On the other hand, 'list' at the mechanistic level denotes a specific computational object which has a structure and which is manipulated in various standard ways. The ordinary word 'list' may be relatively meaningless at the logical level, although 'set' might well capture much of the significance of the mechanistic term 'list'.

By paying close attention to the language and the line of reasoning proffered by subjects in protocol sessions, it is possible to locate them at the various discourse levels and to determine the
point of change from one discourse to another. This was illustrated in Chapter 7, where Subject 4 changes from considering how he would go about deciding the eventual outcome of the program to how Prolog would do it. His initial comments refer to what he would do (hence the 'I'):

3 Clause 1 - a(X) if b(X) and c(X),
4 uhm I think I'd probably start
5 and look down at the facts
6 to see what might be instantiated.

But then he changes his point of view, and considers Prolog's execution strategy (note the change from 'I' to 'it' - Prolog):

16 Well its going to set up sub-
17 ...suppose we give it the goal ?-a(X)
18 it'll set up the subgoal b(X)
19 and in trying to satisfy that it will take clause 2 here
20 and then try and satisfy another subgoal d(X)....

This protocol was contrasted with another subject who failed to make the distinction in the protocol session between what she would do and what Prolog would do. This led her to generate an explanation which did not map properly into the mechanistic domain because concepts of common-sense (derived from the general level) and the notion of reductio ad absurdum (from the logical level) were used as the tacit basis of explanation. As a result, Prolog was inappropriately imbued with a form of forward-looking goal directedness.

10.4 Superstrategies and Superbugs

Figure 10.2 is a table of the superstrategies and superbugs identified in the study and the numbers of students who exhibited them. Not all students were led into error by these strategies. It should be remembered that the intention of the
Fig. 10.2 Superstrategies and Superbugs Identified in the Study

observational studies was not simply to demonstrate the frequency of bugs or errors, but to examine what happens when beginners are presented with Prolog, how they interpret it and how they try to use it. To this end, only the best examples of particular difficulties were selected for discussion, and in some cases the discussion was speculative with regard to potential for such errors based on experiences of teaching students rather than protocol sessions (e.g. as in the Left to Right bias superbug). The value of the studies, then, lies more in being able to interpret and comment upon overall strategies used by novices rather than in demonstrating that some percentage of students did or did not demonstrate a particular error.

Nevertheless, during observation, bugs were noted, particularly in the MSc study. These are shown in Figure 10.3.
Again, tabular presentation may be less than satisfactory in that most of the interesting information is left out. For example, recall that the single occurrence of 'Try once and pass and cut' and 'Redo body from the left and cut' were in two individuals who did not show this bug in their first protocols (i.e. the ordinary TOAP and RBFL). In other words, the context of the protocol task led these two to make this error, rather than their having, for example, a 'buggy' mental model of backtracking in general. This point will be taken up again later.

10.5 Future Work

The research reported here has not had the advantage of being based on a well-developed existing theory of novice programming behaviour, since existing theories do not account for, or explain, the sorts of behaviour and misconceptions observed in the students in this study. A major problem has been, therefore, on the one hand to provide opportunities for observing novice Prolog programmers in reasonably realistic non-stressful situations, (i.e. 'learning' ones rather than 'experimental' ones), whilst at the same time to
devise test materials which are both exploratory and yet sufficiently well tuned to yield substantial evidence of superstrategies and potential superbugs. This is clearly a cyclic development, and the thesis represents a first step in the process. The three-levelled discourse framework imposes a particular view of the novice and the task of learning to program, and its use has been shown to yield a coherent account of certain sorts of novice behaviour.

However, the research is still at an inductive stage, and the interpretation needs now to be subjected to a more thoroughgoing and rigorous analysis. Future steps are clear.

**Repeating Studies and Refining Test Materials**

The most obvious is to repeat the studies using a much broader population of learners than was possible in this study. For example, since a major strand of the argument in the thesis focuses around students' misinterpretations of formal domains, it would be interesting to see whether students who have strong logic or mathematics backgrounds suffer from the same types of difficulties. Studies which compared performance between groups of students with backgrounds in other formal domains might establish whether or not such training eradicates the worst effects of superbugs. On the grounds that Alex - who had worked with computers and computing languages before - experienced a great many conceptual difficulties with Prolog, it seems unlikely that they would have a trouble free ride. For example, it may be that students acquainted with formal domains may not suffer from the natural language superbug, but, like the MSc students, may well have misapprehensions about the
nature of the discourse with a computer. Or they may have difficulties because the particular prior knowledge they have (and bring to bear) may lead to violated expectations in just same way Alex's experience with computers led him into difficulties.

The test materials also need development and fine tuning. For example, in retrospect, the instrument used to investigate causal reasoning in Prolog was the equivalent of a psychological sledgehammer (i.e. including a 'because' in an expression). A great deal of interesting data about Prolog being used to represent causal reasoning may have been lost by the confrontational nature of this task. As this was the only group of Computers and Thought students who were taught Prolog there was no opportunity to repeat the exercise with a similar subject pool. In order to devise more subtle methods for investigating this issue, an analysis of causal reasoning in natural language and explanation would be required, an analysis beyond the scope of the present work.

**Context Dependency and Backtracking**

It would also be interesting to follow up in more detail apparently correct performance in novices. Opportunities did not arise during these empirical studies to deeply probe the understanding of students who appeared to be competent Prolog programmers and who did not volunteer to participate in studies. But of those MSc students who did volunteer (not surprisingly) very few had a complete and robust understanding of what was going on, although most went on to become proficient (even excellent) Prolog programmers. There was a sense of developing a 'good-enough' understanding to keep progressing.
Two questions arise from this:

(i) are certain types of apparent misapprehension (such as Try-Once-and-Pass and Redo-Body-from-the-Left) really harmful, or do they simply represent a necessary phase in learning which lays the ground for the future development of a robust interpretation?

(ii) are such errors context dependent (i.e. one program may lead an individual to make TOAP, whilst another would not).

In other words would a student who showed Try-Once-and-Pass in one session consistently produce the same error in another session, or with another example program? RBFL and TOAP may only arise because students have never had to provide so detailed an account of execution before and are therefore generating an explanation on the fly, so to speak. Generating a correct and complete explanation from scratch and making a few mistakes in the process is a very different case from having deep misconceptions about backtracking.

To investigate question (ii), a study could be undertaken in which students are asked firstly to described what they thought a program (of the Coombs and Stell type) was supposed to do, as they were in the studies reported here. But then, they would be presented with traces of the execution and asked to identify the one which fits their description (either Byrd box model traces, or pictorial representations). Such traces might be modified to represent TOAP and RBFL as well as the correct trace. This would put students in the interesting situation of being able to compare their own original interpretation with a collection of others, only one of which would in fact be a 'true' trace.

A pilot study has been run along these lines (Hook, Taylor and du Boulay, unpublished draft), and the preliminary observations of the as yet unanalysed data are that students immediately begin
reasoning about the traces, comparing them with their own interpretation, and begin to correct themselves straightaway. In this case, students who showed RBFL or TOAP in their verbal account seem to recover rapidly when faced with a correct trace.

This manipulation of the Coombs and Stell test may therefore provide further information on high level strategies which enable a student to recognise the correct trace even if it does not correspond with her own initial interpretation.

To address question (ii) above, a further extension of the study would then be to ask experts to perform the same task, comparing their ability to generate the actual execution trace with that of comparative beginners, and the speed with which they recognise the correct trace. Observing whether or not TOAP and RBFL occurred in experts would also provide information as to the longevity of the bugs. Furthermore, their existence in expert reports would imply that they may not be very harmful.

**Visual Programming for Prolog**

Since the main argument throughout the thesis has been that ordinary discourse processes and natural language understanding are used to interpret programming tasks, possible future research projects might involve devising ways of presenting Prolog to learners which do not rely heavily on linguistic interpretation, and which may help short-circuit some of the more detrimental effects of general discourse on programming.

It has been argued in this thesis that, in the early phases of learning to program, beginners interpret Prolog clauses using the
semantics of natural language. This tendency is compounded by the fact that novice programmers are inclined to tacitly assume that the computer is a coherent, inferencing 'other'. In this case, the temptation is for beginners to write programs comprised of sets of instructions, in English, which make sense to them. This not only leads to incomplete specifications, but also leads to programs which cannot be interpreted by the compiler to produce the desired behaviour.

Bonar and Cunningham (1986) address the issue of moving from beginners' natural language solutions to working code in their BRIDGE system which supports the evolution from students' initial descriptions of problems in English through to Pascal code structures. By the provision of a number of different pre-packaged representations of the original problem, students are guided through a process of successive approximation from their own proposed solution to a complete program by selecting goals and mapping them onto existing plans in BRIDGE's library. These plans are then instantiated with chunks of code selected from a menu by the student.

This method seems promising, and provides a link between the programmer's initial plan, and a running program. Adopting this approach with Prolog is not straightforward though, because Prolog does not have the typical standard structures (e.g. different types of loop constructs) which can easily be stored in form of plans in a library (Fung, 1987). This is due in part to the use of unification, where instantiations of variables form the major part of the computation.
From the foregoing discussion, it seems crucial to find ways of representing Prolog programs which does not limit the learner to 'sets of instructions' expressed in textual form which are susceptible to misinterpretation.

In order to help beginners (and experts) in the development of Prolog programs, the Transparent Prolog Machine (Eisenstadt and Brayshaw, 1986) - hereafter TPM - uses the technique of program visualisation (Myers, 1986). The program is specified in the conventional, textual manner and graphics are used to illustrate the program and its run-time execution. TPM thus employs dynamic program visualisation using AORTA diagrams (Augmented AND/OR Trees).

An interesting project would be investigate the possibility of converting this basic system from program visualisation to visual programming. That is, instead of viewing the TPM as an aid to understanding already written programs, AORTA diagrams could be used as a means for teaching students to write Prolog programs: a Prolog program could then become, not a set of clauses, but a diagram. A system which converts program plans represented in the diagrammatic form of TPM into Prolog clauses, and, therefore, into a running program, would need to be constructed.

The advantage of this approach is that rather than having to re-represent the diagrammatic solution in the particular constructs of the programming language, the diagram is the program. Having represented the plan diagrammatically, the student would then be able to watch its execution in ways which are already available with TPM. This, in principle, could provide a concrete link between
initial program conceptualisation and planning, and the execution model of Prolog.

To achieve the goal outlined above, empirical research would need to be directed at two particular issues. The first would be how novices can be assisted in converting their initial outline solutions to problems into diagrammatic form; and the second is effective ways of helping beginners come to terms with unification in such a way that rather than using TPM to understand how clauses such as 'append' work, learners could find themselves able to generate the 'append' clauses spontaneously for themselves.

The first step in this research would be to investigate the feasibility of the mapping between declarations and the diagrams used in TPM. Empirical studies should address two issues. The first is whether novices are able to understand given Prolog programs represented as AND/OR trees, and can map between Prolog clauses and such trees. This is what learners are expected to do at present when using the TPM. The second is whether they can accomplish the more difficult task of mapping their own program plans into AND/OR trees for themselves. These two tasks represent the crucial difference between program visualisation (i.e. program comprehension) and visual programming (i.e. program generation).

Both these studies could be accomplished using pen and paper materials, and could therefore be carried out on a variety of students at different stages in learning the language. After investigations of novices, the use of these tools by more advanced programmers would demonstrate how far visual programming techniques can be taken as expertise increases. For example, it may be that
the diagrammatic representation is sufficiently robust to be used as the basis for a logic programming software design tool. Alternatively, because it is specifically aimed at novices in the initial instance, it may turn out to be too cumbersome to use for more sophisticated programs.

Models for Unification

A further difficult area for novice Prolog programmers, and one which has not been discussed in detail in this thesis, is the role that unification plays in Prolog programming. Unification is problematic to represent effectively, and none of the available teaching models directly address this issue (e.g. the TPM, Byrd Boxes etc.).

There are two possible lines of research to pursue. The first would develop a tracer based upon an AND/OR tree, running on a graphics workstation, which has a window specifically dedicated to showing the current state of datastructures with variables in them as execution proceeds. This display would be uncluttered by any other information. Thus, the programmer could simply observe the effect of forward execution and backtracking on the datastructures - and nothing else. This facility would be relatively cheap and simple to construct and could be added to existing trace packages.

A second more radical approach would be to develop a range of concrete metaphors for the unification process. Such a project is more ambitious and would require a great deal of careful empirical evaluation studies to identify a sufficiently powerful metaphor (or collection of metaphors) which is flexible enough, for example, to eventually enable a student to understand unification in complex
cases such as 'append'. This is a hard task: there is a fine balance between manipulating an analogy in such a way that it still makes sense, and, in ad hoc fashion, forcing a fit.
10.6 Afterword

Superstrategy and superbug analysis has provided a great deal of information with regard to what novices do when they are faced with the difficult task of learning to program. The data which such a perspective yields is both rich and complex, and, under certain circumstances, intractable. An attitude sometimes encountered in teachers is that there are as many programming misconceptions as there are learners of programming, and certainly, a great deal more data illustrating idiosyncratic programming 'style' was collected in the studies than could be used in the thesis. But nevertheless, as Elshout et al. (1986) speculate, there does seem to be some generality in novice performance which is not confined to a specific problem domain. The strategies which have been discussed are similar to those observed in the deductive reasoning literature, and also with those reported by Elshout in relation to physics novices.

Whilst studies of the sort conducted here can be methodologically problematic, they do provide the basis for developing new theoretical perspectives which address the complexity of the task of programming rather than trying to strip it out. Such analyses also force us to acknowledge the flexibility, originality and success of human problem solving, as well as its failings. We may also wonder in amazement that so many people who co-operate as willing subjects can do so much with so little in the way of support.
APPENDIX 1

Texts of Problem Statements

The Himalayan Tea Ceremony

In the inns of certain Himalayan villages is practised a most civilized and refined tea ceremony. The ceremony involves a host and exactly two guests, neither more nor less. When his guests have arrived and have seated themselves at his table, the host performs five services for them. These services are listed below in the order of the nobility which the Himalayans attribute to them:

- Stoking the Fire
- Fanning the Flames
- Passing the Rice Cakes
- Pouring the Tea
- Reciting Poetry

During the ceremony, any of those present may ask another, "Honored Sir, may I perform this onerous task for you?". However, a person may request of another only the least noble of the tasks which the other is performing. Further, if a person is performing any tasks, then he may not request a task which is nobler than the least noble task he is already performing. Custom requires that by the time the tea ceremony is over, all of the tasks will have been transferred from the host to the most senior of the guests. How may this be accomplished?

The Towers of Hanoi

Somewhere near Hanoi there is a monastery whose monks devote their lives to a very important task. In their courtyard are three tall posts. On these posts is a set of sixty-four disks, each with a hole in the centre, and each of a different radius. When the monastery was established, all of the disks were on one of the posts, each disk resting on the one just bigger than it. The monks' task is to move all of the disks to one of the other pegs. Only a single disk may be moved at a time, and all the other disks must be on one of the pegs. In addition, at no time during the process may a disk be placed on top of a disk that is smaller than it. The third peg can, of course, be used as a temporary resting place for the disks. What is the quickest way for the monks to accomplish their mission?

N. B. Students are normally only expected to solve the problem initially using three or five disks.

The River Problem

A farmer is by a river with a wolf, a goat and a cabbage, and wishes to cross to the other side. There is a boat which will hold only two 'objects' at a time. Only the farmer knows how to row the boat. He must move all three items from the left to the right bank without leaving the wolf alone with the goat or the goat with the cabbage. There are obvious gastronomic reasons for this
stipulation. Note that the wolf is not particularly partial to cabbage, and may be left alone with it.
APPENDIX 2

BRIEF OUTLINES OF CURRICULUM MATERIALS

The following sections outline the Computers and Thought Course and the MSc. course. The course handouts provided for students are reproduced, along with selected sections of TEACH files which are part of the POPLOG system at Sussex University. Sections of teach files are reproduced by kind permission of their authors.

1. COMPUTERS AND THOUGHT

1.1 Course Handout

Introduction

The Computers and Thought course aims to introduce some general topics, teach some useful skills and act as taster for later courses.

The general topic is Artificial Intelligence, which concerns computer simulations of human abilities, like mental imagery, problem solving or understanding English. The skill is in computing - writing programs in Prolog, an advanced but friendly programming system which includes text processing facilities. The later courses are those in the Cognitive Studies Programme, and the Computer Models of Mind contextual in the School of Social Sciences.

Computers and Thought is relevant also to students who plan to take a course on Philosophy of Mind, or courses on Psychology or Linguistics. Besides its relevance to specific courses available at Sussex, it introduces and illustrates concepts and techniques which will help you to appreciate the potential uses of computers in a wide variety of applications, in offices, in the home, in schools, in hospitals, etc.

In addition it should help you think about what you are. Are you some kind of machine, with programs governing your behaviour? If not, how do you differ from a programmed computer? In order to have sensible views on such questions you need to know more about computers than you can learn from the popular press, or from programming in BASIC on a PET or APPLE or similar microcomputer.

The course is taught by means of a combination of lectures, tutorials, and practical programming exercises.

You will be allocated to a tutorial group with one of the course tutors and you will have a weekly tutorial of one hour in a group of four or five. There will also be opportunities for individual consultation. For tutorial times see the notice on the School of Social Sciences first year notice board, opposite room E.325. It is important that you select a time for tutorials on that notice.
COURSE STRUCTURE

The course will use handouts like this one, and 'help files' or 'teach files' stored on the computer. Most contain programming exercises to be done on the computer. In addition to regular programming exercises, discussed in tutorials, the last week or two of the term will be spent on a programming project.

Programming exercises include writing reports on your programs. We place great importance on the ability not only to write programs but also to explain clearly what their purpose is and how they work.

WEEKS 1 and 2

Get acquainted with the computer (see WEEK1.CT). Learn to log in and out. Play with some demonstration programs, including CHAT a program that answers questions about the geography of the world. Learn to use the 'VED' editor for simple "word processing" and the 'MAIL' and 'DIRECTORY' commands.

WEEKS 3 and 4

Learn a simple notation for expressing facts and rules that an intelligent system might need to know. Do some exercises with this notation and hand in a short report. Investigate using this notation as a "programming language" (called Prolog) for the computer (TEACH VEDPROLOG). Hand in an essay (see ESSAY) for discussion in week 4.

WEEKS 5 and 6

Continue with exercises involving the writing of more complex Prolog programs (TEACH RULES and TEACH FAMILY). Understand in more detail how Prolog actually works. Write a brief report on the programs you have written. Write a brief report on programs you have written.

WEEK 7

Start working on SIR, a simple program that can be told facts and answer questions in English.

WEEKS 8 to 10

Continue with SIR and/or branch out into other programming exercises in consultation with your tutor. Choose one of them as a PROJECT. Choose a project of your own design or do one of the following:

A: TURTLE. Experiment with the TURTLE program to draw pictures. Produce programs specifying how complex pictures can be constructed out of simpler ones.

OR:

B: PARSING. Experiment with the GRAMMAR program to see how the
structure of sentences can be described in a more sophisticated way by STIR-like patterns. Design a grammar for a subset of a natural language and use the program to examine the syntactic structure of some example sentences.

OR:

C: RIVER. experiment with the RIVER program which models the behaviour of a "micro World". Write a similar program of your own for another "micro World".

The report on the program should be submitted to your tutor by the Friday of WEEK 9.

WEEK 10

Final meeting. Get back your project reports with tutor's comments. Discuss how the course has gone and what it all adds up to.

The lectures will in part introduce you to basic concepts and techniques needed for your programming activities. In addition they will tell you about some of the achievements of researchers in Artificial Intelligence, e.g. in language, vision, expert-systems, education and so on. Finally, they will raise the question whether human beings should be thought of as computers (albeit very sophisticated ones) and the question whether machines can be made to think, have emotions, etc. There may also be some discussion, in lectures or tutorials of the social implications of the computer revolution and work in Artificial Intelligence.
### Summary Table for Lectures, Seminars and Teach Files

<table>
<thead>
<tr>
<th>WEEK</th>
<th>TUESDAY LECTURE</th>
<th>WEDNESDAY LECTURE</th>
<th>THUR/FRI SEMINAR</th>
<th>TUTORIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Computers</td>
<td>Computers</td>
<td>log in</td>
<td>log in</td>
</tr>
<tr>
<td></td>
<td>logging in</td>
<td>logging in</td>
<td>set WEEK1.CT</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CHAT</td>
<td>Language</td>
<td>Language</td>
<td>WEEK1.CT</td>
</tr>
<tr>
<td>3</td>
<td>VED</td>
<td>Problem-solving</td>
<td>Prolog intro</td>
<td>VED/CHAT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>facts,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>questions</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Prolog</td>
<td>Vision</td>
<td>Vision</td>
<td>Essay</td>
</tr>
<tr>
<td></td>
<td>VEDPROLOG</td>
<td>Vision</td>
<td>Vision</td>
<td>Essay</td>
</tr>
<tr>
<td></td>
<td>variables</td>
<td>Vision</td>
<td>Vision</td>
<td>Essay</td>
</tr>
<tr>
<td>5</td>
<td>rules</td>
<td>Speech</td>
<td>Search</td>
<td>VEDPROLOG</td>
</tr>
<tr>
<td>6</td>
<td>unification</td>
<td>Expert-systems</td>
<td>Expert-systems</td>
<td>RULES</td>
</tr>
<tr>
<td></td>
<td>lists</td>
<td></td>
<td></td>
<td>FAMILY</td>
</tr>
<tr>
<td>7</td>
<td>introduce</td>
<td>Education</td>
<td>MATCHING/</td>
<td>Plan</td>
</tr>
<tr>
<td></td>
<td>projects SIR,</td>
<td></td>
<td>TRACER</td>
<td>project</td>
</tr>
<tr>
<td></td>
<td>TURTLE, RIVER</td>
<td></td>
<td></td>
<td>SIR, TURTLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RIVER</td>
</tr>
<tr>
<td>8</td>
<td>SIR</td>
<td>Computers</td>
<td>Computers</td>
<td>MATCHING</td>
</tr>
<tr>
<td></td>
<td>microworlds</td>
<td>&amp; thought</td>
<td>&amp; thought</td>
<td>TRACER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Help</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Project</td>
</tr>
<tr>
<td>9</td>
<td>backtracking</td>
<td>Philosophy</td>
<td>recursion</td>
<td>Help</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>project</td>
</tr>
<tr>
<td>10</td>
<td>hardware</td>
<td>Social</td>
<td>Debate</td>
<td>Assess</td>
</tr>
<tr>
<td></td>
<td>5th generation</td>
<td>Implications</td>
<td>Social</td>
<td>project</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>implications</td>
<td></td>
</tr>
</tbody>
</table>
1.3 TEACH Files for Computers and Thought

These sections represent, where possible, the introduction followed by the contents list of each file. They are intended to provide an illustration of the sorts of tasks Computers and Thought students would be expected to accomplish and do not represent the full course.

TEACH VEDPROLOG

This TEACH file shows you how to create program files for PROLOG using VED, and how to write very simple programs which will answer questions. You will be given information about the programming language PROLOG - e.g. PROLOG facts, questions, and variables - and about how to use the editor (VED) to create program files. Most sections have a header which indicates whether it is PROLOG or VED that is being referred to.

INDEX FOR THIS FILE:

- CREATING YOUR OWN PROGRAM FILE IN VED
- SWITCHING BETWEEN THREE FILES WITH <ESC>-e
- PROLOG: OBJECTS AND RELATIONSHIPS
- CREATING A PROLOG PROGRAM
- ADDING THE PROGRAM TO THE PROLOG DATABASE
- ONE WAY TO LOAD A FILE:
- MISHAP MESSAGES
- VED OUTPUT FILES - OR WHERE TO HAVE A DIALOGUE
- HOW TO ASK QUESTIONS IN PROLOG
- PROLOG VARIABLES
- WHAT IF THERE IS MORE THAN ONE SOLUTION?
- CONJUNCTIONS
- DEFINING NEW RELATIONS
- WRITING YOUR OWN VERSION OF CHAT
- SUMMARY OF PROLOG SO FAR
- SOME TERMINOLOGY
- SUMMARY OF VED SO FAR

.

TEACH RULES

This TEACH file shows you how to incorporate rules into a PROLOG program to make it more complex. You should already have looked at TEACH VED and TEACH VEDPROLOG before attempting this file.

In TEACH VEDPROLOG you were introduced to PROLOG FACTS, which are statements about RELATIONS between OBJECTS. You also learned a little bit about VARIABLES, and how to use variables in questions. This file will teach you how to write PROLOG RULES, and how to use variables in the rules. This will allow you to write more complex programs.
INDEX FOR THIS FILE:

-- RECAPITULATION
-- RULES IN PROLOG
-- RULES WITH CONJUNCTIONS
-- GOALS AND SUB-GOALS
-- SOME TERMINOLOGY -- CLAUSES
-- WHAT TO DO NEXT

FAMILY TREE

This TEACH file tells you how to construct a database of family relationships. The database will contain FACTS and RULES, and you will be able to find out about relationships by asking QUESTIONS which contain VARIABLES. For example, you could ask:

Who is the mother of William II?  ?- mother(X, william2).
Who is a brother of William II?    ?- brother(X, william2).

and more complex questions which will be discussed later.

You should already have looked at TEACH VEDPROLOG and TEACH RULES before trying these exercises. In TEACH VEDPROLOG you will have met PROLOG FACTS and VARIABLES. TEACH RULES introduces you to PROLOG RULES. In this file, we elaborate on the use of rules and variables.

INDEX FOR THIS FILE:

-- FAMILY TREES
-- SOME REVISION AND NEW TERMINOLOGY
-- FAMILIES IN PROLOG
-- THREE PLACE PREDICATES
-- ASKING QUESTIONS
-- MAKING SIMPLE RULES
-- USING 'NOT'
-- BUILDING WITH FACTS
-- USING RULES TO BUILD MORE RULES
-- MORE AND MORE RELATIVES!

TEACH SIR

This TEACH file tells you how to construct a simple program which can converse in English and also "learn" new information from what it is told. It is based loosely on the following program:

Bertram Raphael
SIR: A COMPUTER PROGRAM FOR SEMANTIC INFORMATION RETRIEVAL
in SEMANTIC INFORMATION PROCESSING (ed M. Minsky, MIT Press, 1967)

INDEX FOR THIS FILE:

-- INTRODUCTION
-- SENTENCE PATTERNS
-- USING LISTS TO REPRESENT SENTENCES
TEACH TURTLE

This file introduces the Prolog 'turtle' library package, which enables you to draw pictures on a rectangular grid, using the procedures DRAW, JUMP, TURN, DRAWTO, JUMPTO, and more. You should be familiar with TEACH BUFFERS before you try this teach file.

The turtle package provides a number of "built-in" predicates for drawing. A goal involving one of these predicates ALWAYS succeeds, but satisfying such a goal causes an "external event" to take place, such as a line being drawn on the screen. For drawing pictures it matters what order various lines are drawn. In order to create and understand turtle programs, therefore, it is essential to understand how Prolog works in enough detail to know in what sequence the various goals will be tackled.

INDEX FOR THIS FILE:

-- GETTING THE TURTLE PROGRAM READY
-- MAKING THE TURTLE FACE IN DIFFERENT DIRECTIONS
-- DEFINING NEW PREDICATES
-- CHANGING THE PAINT
-- SAVING YOUR PICTURE
-- REVISION

........

TEACH CHAT

CHAT is a program for interrogating in natural language a database of facts about the countries of the world. (Copyright (C) 1982 David Warren, SRI International, 333 Ravenswood Ave., Menlo Park, California 94025, USA; Fernando Pereira, Dept. of Architecture, University of Edinburgh, 20 Chambers St., Edinburgh EH1 1JZ, Scotland).

The program expects you to type in an English query (some examples are given below). A query must end with a '.' or a '?'. It can extend over several lines if necessary. Here are some examples of sentences that CHAT may be able to understand:

What rivers are there?
Does Afghanistan border China?
........
2. MSc COURSE

2.1. Lecture topics

Week 1:

1. Introduction, and
   Prolog - Background I: AI Programming
2. Prolog - Background II: Logic

Week 2:

3. Anatomy of Prolog I: Syntax
4. Anatomy of Prolog II: Execution

Week 3:

5. List processing in Prolog
6. Problem solving with Prolog I: Route finding

Week 4:

7. Problem solving with Prolog II: The river problem
8. Prolog programming schemas and programming style.
9. Backtracking and "Cut".

Week 5:

10. The Prolog database, and miscellaneous features.
11. "Setof", and implementing "findall".

Week 6:

13. Definite Clause Grammars

Week 7:

14. Planning with Prolog: PSTRIPS
15. Production System programming.

Week 8:

16. A Prolog interpreter in Prolog
17. Prolog and POP-11: Multi-language programming.

Week 9:

18. Object oriented programming I: Flavours
19. Object oriented programming II: Frames-based programming
2.2 TEACH Files

MSc TEACH files are often created for specific exercises, and are not as elaborate as those for Computers and Thought. In this section, the introduction is provided as well as the contents lists (where possible) and then some idea is given of the sorts of questions and exercises which students are expected to be able to cope with after they have finished working through the file. These are not complete texts of TEACH files.

TEACH SOCCER

A Simple PROLOG Exercise.

The following PROLOG clause

score(week(1), everton, 4, newcastle, 0).

records the information that everton beat newcastle at home in the first week of the season, and that the score was 4 - nil.

Here are some more football results. Record them in PROLOG, using a similar format.

WEEK 1

leicester 0 stoke 0
man utd 0 coventry 1
norwich 1 southampton 0
....

WEEK 2

arsenal 2 coventry 1
everton 4 watford 0
leicester 1 chelsea 1
....

Some queries.

To find out what everton scored in week 2 you could ask

The response will be

Homescore = 4

But this presupposes that you know that everton played at home. Here is a predicate 'teamscore' for getting the score of a team for a given week where you don't know whether they played at home or
away. It is defined with two clauses.

teamscore(Week, Team, Score) :-
    score(Week, Team, Score, _, _).

teamscore(Week, Team, Score) :-
    score(Week, _, _, Team, Score).

Explain how this might be used, with a couple of examples.

How might you complete the following definitions?

home(Team, Week) :- .......
away(Team, Week) :- .......
won(Team, Week) :- .......
(Use the '>' operator in this one.)
lost(Team, Week) :- .......
beat(Winners,Losers, Week) :- .......
drew(Team1, Team2, Week) :- .......

This last one can be done in two ways: In the first case you should include a condition such as 'Score1 = Score2'. The second, more economical, way avoids the need for bringing in the '=' sign.

The following facts give lists of teams by division:

division(1, [everton, newcastle, leicester, stoke, man utd, coventry, norwich, southampton, arsenal, ipswich, aston_villa, watford, chelsea] ).

division(2, [cardiff, barnsley, grimsby, blackburn, huddersfield, man_city, notts_co, shef utd, middlesboro, wimbledon] ).

division(3, [bolton, derby, bournemouth, doncaster, hull, bradford, wigan, millwall, bristol_r, cambridge] ).

Here's a way of saying which division a given team is in, using the 'member' predicate.

divteam(Team, D) :-
    division(D, List),
    member(Team,List).

Try using 'divteam' on a few examples. Call the goal with the first, then the second, then both arguments uninstantiated.

Define a predicate 'canplay(Team1,Team2)' which will be satisfied by any two teams in the same division. (You can use 'divteam' as
defined above, or alternatively just use 'division' and 'member' - try both ways.)

Invent some more predicates of your own choosing.

TEACH BLOCKEX

Here is an arrangement of eight blocks of different colours:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>b2</td>
<td>b3</td>
<td>b4</td>
</tr>
<tr>
<td>[red]</td>
<td>[green]</td>
<td>[blue]</td>
<td>[red]</td>
</tr>
</tbody>
</table>

---

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b5</td>
<td>b6</td>
<td>b7</td>
<td>b8</td>
</tr>
<tr>
<td>[green]</td>
<td>[purple]</td>
<td>[blue]</td>
<td>[yellow]</td>
</tr>
</tbody>
</table>

COL. 1    COL. 2    COL. 3    COL. 4

The arrangement of the locations of each block can be expressed in PROLOG facts of the following format:

location(b3, 1, 3)

etc.

Colours of blocks could be expressed in the following way:

colour(b1, red).

etc.

Fill in a database of block locations and colours.

Define predicates as follows:

row(Block,R):- ....

(so that, e.g., 'row(b5,2).' succeeds).

column(Block,R):- ....

samerow(B1,B2):- ...

samecol(B1,B2):- ...

Here is a definition of 'justleftof':

justleftof(B1,B2):-

samerow(B1,B2),

column(B1,C1),

column(B2,C2),

C1 is C2 - 1.

Notes: (1) This assumes that 'samerow' is already defined.

(2) The final condition uses the built-in PROLOG operator 'is'.

This takes the term 'C2 - 1', and evaluates it as an arithmetical expression, and then lets C1 stand for the result. (See Clocksin & Mellish 2nd ed, p. 36)

Define 'justrightof' in terms of 'justleftof'.

Define 'leftof' in such a way that it holds between two blocks in the same row if the first is somewhere on the left of the second. Define 'rightof' in terms of 'leftof'.

Define 'justover' and 'justunder' in an analogous way to 'justleftof' and 'justrightof'. Make it so that it would apply if there were more than two rows of blocks.

Define 'over' and 'under' in analogous ways.

Define 'adjacent(B1,B2)' which will be satisfied by two blocks which are horizontally or vertically next to each other. Modify it to allow for diagonally adjacent positions.

How about a definition of a relation between two blocks which are on a diagonal line with each other, but not necessarily on diagonally adjacent positions.

Add further spatial definitions as you wish. Introduce further blocks.

......

TEACH LISTEX

Some exercises with lists.

1. Using (a) a tree diagram; 
   (b) a box diagram,

   show how each of the following list structures could be represented as dotted pairs:

   \[
   \begin{align*}
   &[a,b,c] & [a,b] & [a] & [a,[b,c]] \\
   &[[a],[b,[c]]] & [a|X] & [a,b|X] & [[A|B]|[C|D]] \\
   \end{align*}
   \[

   2. What are the heads and tails (in the strict sense of "head" and "tail") of the following list-structures?

   \[
   \begin{align*}
   &[a,b,c,d] & [a] & [a,[]] & [a,b,[c]] \\
   &[[a],[b,c]] & [[a,b],[c]] & [a | X] & [a,b | X] \\
   &[] & [[]] & [[]][] & [[]][] \\
   &[[a]] & [[a][X]][[Y][Z]] & [a | [b,c]] & [a|[b|[c|[d|]])] \\
   \end{align*}
   \[

   6. Write a program that will change:
Define a predicate 'change' which will swap individual words, (for instance the definition should include the clause 'change(pavement, sidewalk)') and a predicate 'alter' that will alter a list of words, by changing individual ones.

Write a (non-recursive) version of alter which will only work on lists of the form '[the, X, was, Y]'.

Modify 'alter' so that it will work on lists of any structure. This version will have to be recursive. You will need a 'bottoming-out' clause for 'alter' that deals with altering an empty list. You'll also need a clause which says that List1 can be 'altered' to List2 if the head of List1 can be 'changed' to the head of List2, and if the tail of List1 can be 'altered' to the tail of List2.

10. Write a program to solve the Towers of Hanoi problem for four discs, by searching through a state-transition space, where each node is defined as a list specifying membership of the start-peg, the destination peg, and the spare peg, respectively. Thus

startstate ( [ [4,3,2,1], [], [] ] ).
goalstate ( [ [], [4,3,2,1], [] ] ).

TEACH CROSSRIVER

NB: this teach file follows on from the lectures about "Search and Problem Solving in Prolog", covering and extending of the material presented.

CONTENTS:

- THE ROUTE-FINDING PREDICATE
- GRAPH SEARCHING AND PROBLEM SOLVING
- A NEW DOMAIN - THE RIVER PROBLEM
- STATE REPRESENTATION
- PROBLEM SPACE
- PROLOG DESCRIPTION OF THE PROBLEM SPACE - VERSION 1
- FURTHER DESCRIPTION OF THE PROBLEM
- RUNNING VERSION 1 OF THE PROGRAM
- A STATE-GENERATING VERSION OF THE PROGRAM
- REPRESENTING LEGALITY OF STATE TRANSITIONS
- DEFINITION OF LEGAL/2
- DEFINITION OF OBJECT/4
- REPRESENTING SAFETY OF STATES
This teach file will use the route-finding predicate developed in TEACH * ROUTE. It is assumed that you understand the principles on which it works, namely Prolog's built-in search through backtracking.

**TEACH GRAMMAR RULES**

**CONTENTS**

- INTRODUCTION
- GRAMMAR RULES
- SYNTACTIC SUGAR
- FROM TREES TO CLAUSES
- EXERCISES

**Introduction**

This file is concerned with how one might generate executable PROLOG clauses from PROLOG grammar rules. The description of what a grammar rule is and how it can be used is simply intended to remind people who have already come across them already of some of the details. The main function of this file is to explain how to turn a grammar rule into a clause.

**Exercises**

1. Use the above program to convert

   
   sentence --> noun phrase, verb phrase
   noun_phrase --> determiner, noun.
   verb_phrase --> verb, noun_phrase.

   into a set of applicable clauses. Add the additional clauses

   
   determiner([the | Rest], Rest).
   noun([girl | Rest], Rest).
   noun([bike | Rest], Rest).
   verb([fixed | Rest], Rest).

   and spy what happens when you call

   
   sentence([the, girl, fixed, the, bike], Remainder).

2. Convince yourself that you understand why the dictionary entries have the form

   
   category([word, Rest], Rest).

   Write a predicate LEXITEM which will assert the appropriate rule when called as

   
   lexitem(girl, noun).
(3) Adapt MAKE RULE so that it will create appropriate clauses for the general form of direct clause grammar, where the rules may already have extra variables, e.g.

\[ S(\text{Trans}) \rightarrow \text{NP(}\text{Number}\text{)}, \text{VP(}\text{Number, Trans}). \]

(This is quite hard).

(4) Write a version of MAKE RULE which will construct clauses to apply the grammar rules bottom up. (This is quite hard).

**TEACH PSTRIPS**

--- CONTENTS
--- BACKGROUND
--- REPRESENTATIONS
--- VARIABLES AND OPERATOR SCHEMAS
--- IMPLEMENTATION IN PROLOG
--- EXERCISES
--- REFERENCES

Background

This file indicates how you might write a simple means-end (goal reducing, backward chaining) problem solver in PROLOG. A lot of difficult problems with such programs will be left open - see TEACH STRIPS and TEACH SOLVER for a more sophisticated (but almost ..... [A program of the following form is developed:]

(i) \( \text{plan(}\text{World, Goals, [], World}) ::- \) not(next untrue goal(Goals, _, World)), !.

(ii) \( \text{plan(}\text{World, Goals, Plan, New_world}) ::- \) next untrue goal(Goals, Goal, World), achieves(operator(Op, Preconditions, Changes), Goal), plan(World, Preconditions, Sub_plan, Intermed_world), make_changes(Changes, Intermed_world, Next_world), plan(Next_world, Goals, Rest_plan, New_world), append(Sub_plan, [Op | Rest_plan], Plan).

The cut in clause (i) is for efficiency rather than to affect the overall behaviour of the program. If we have checked that NEXT_UNTRUE_GOAL is false, there is no possible point in going into clause (i), which will immediately try to prove NEXT_UNTRUE_GOAL and will inevitably fail.

This final version of the program works if you have simple operators, simple initial worlds, and simple sets of goals. It needs to be substantially extended in a variety of ways if it is to be any use in more complex situations. The exercises below outline some of the things that you might want to do to improve it. It would not be surprising if some of these required radical reimplementation of the basic algorithm, but some at least should be possible simply by extending it in reasonably straightforward ways.

**Exercises**
(1) Fill in the missing bits in the final version of the planner above (i.e. the predicates NEXT UNTRUE_GOAL, ACHIEVES, MAKE CHANGES). Test your program on a simple set of blocks world operations.

(2) As the program stands, operations are specified in terms of preconditions and effects. It may try to find a plan to achieve any unsatisfied precondition of any operation, even if it is clear (to us) that there is no possibility of finding such a plan (e.g. if one of the preconditions of the operation was that BLOCK(TABLE) should hold, and there are no operations which can possibly turn something into a block unless it is already known to be one). The program may also choose to use some operation to achieve a goal when the goal of that operation is more like a side-effect than a major reason for using it. See if changing the basic representation of operations so that they have the form

\[ \text{operation}(\text{OP}, \text{UNACHIEVABLE PRECONDITIONS}, \text{ACHIEVABLE PRECONDITIONS}, \text{PRIMARY_EFFECTS}, \text{SIDE_EFFECTS}) \]

makes your program perform better (or worse?).

(3) Make your program check that its plans do not contain moves whose effects are undone by later moves. You will have to distinguish between undoing the effects of a move AFTER they have been used (i.e. if move M1 was performed to achieve some precondition of move M2, then it doesn't matter if M2 undoes the effects of M1. But if both M1 and M2 are intended to achieve preconditions of M3, then M2 should not undo the effects of M1).

(4) Make your program check that it is not getting into an infinite loop. It is quite possible for a means-end planner like this to do something like decide it would like to perform action A1; in order to perform action A1, it has to achieve precondition P1; it sees that A2 will achieve this precondition, so it decides to do this; A2 has a precondition P2, which can be achieved by A1, so the program decides to perform A1; in order to do this it needs to achieve P1, which leads it to choose to do A2, which ... Dealing with loops of this sort requires you to keep track not only of what goals are outstanding, but also why you wanted to achieve them in the first place.

(5) Make your program plan hierarchically. A (fairly) simple way to do this is to have levels of preconditions. You start by constructing a plan which achieves all the initial goals using actions whose top level preconditions are satisfied. Once you have this plan, you take each step in it and treat this as a new problem and try to find a plan that achieves this step with all the next level preconditions satisfied, and so on until the lowest level preconditions of all steps in the plan are satisfied.
APPENDIX 3

Protocol Transcripts for Computers and Thought Students:
3-Part Representation Task

Notes:

The Appendix is organised by Subjects rather than by parts of the task. The responses made by each subject have been transcribed literally from their work-paper. These appear first, and are followed by a transcript of the protocol, which represents their comments as they worked. Where students did not comment, the Clause numbers only are provided. So, for example, for Subject 1, Parts 1, 2 and 3 of the task appear consecutively. Since students are engaged in a programming task, the initial letters of the names 'Jaffa' and 'Misha' have been left in lower-case letters, except where students have (erroneously) put them in upper-case in their programs.

COMPUTERS AND THOUGHT: SUBJECT 1

PART 1

RESPONSES:

Clause 1: pencil
Clause 2: Sue is female
Clause 3: Tom is a cat
Clause 4: Mary loves Sue or Sue loves Mary
Clause 5: John loves Henry or Henry loves John
Clause 6: Mary is Sue's mother or Sue is Mary's mother
Clause 7: John is Henry's father or Henry is John's father
Clause 8: Mary and John are Sue's parents
Clause 9: Mary and John are Henry's parents

PROTOCOL:

Clause 1
Clause 2
Clause 3
Clause 4
Clause 5

1 That first one was a bit strange
2 I can't see you could do anything with
3 it you wouldn't get anything like that

E: Right

Clause 6
4 These ones are confusing me now
5 these cards loves
6 because we have just been told that
7 the order is arbitrary
8 of the subject and object

E: Right
9 So I'm putting both options there

E: Right

Clause 7
Clause 8
Clause 9

PART 2

RESPONSES:

Clause 1: don't know
Clause 2: don't know
Clause 3: Student(Jaffa).
Clause 4: Professor(Misha).
Clause 5: Male(Jaffa).
Clause 6: Woman(Misha).
Clause 7: Clever(Misha).
Clause 8: Teaches(Misha, Jaffa). or Teaches(Jaffa, Misha).
Clause 9: Admires(Jaffa, Misha). or Admires(Misha, Jaffa).
Clause 10: Learns well(Jaffa).
Clause 11: Works hard(Jaffa).
Clause 12: Learns well(Jaffa):- Works hard(Jaffa).
Clause 13: Goodstudent(Jaffa):- Admires(Jaffa, Misha),
                  workshard(Jaffa).
Clause 14: Teaches(Misha, X).
                  [then writes]
                  Learns well(X):- Teaches(Misha, X).
Clause 15: SuccessfulStudent(X):- Teaches(Misha, X),
                  Works hard(X).

PROTOCOL:

Clause 1   [writes don't know]
Clause 2   [writes don't know]
Clause 3
Clause 4
Clause 5
Clause 6
Clause 7
Clause 8
Clause 9
Clause 10   [smile]

E: A hard one?

10 [laughs]

E: Which one is it?

11 jaffa learns well
12 well we haven't come across adjectives at all
13 I could do jaffa learns, but not jaffa learns
... Well I suppose you could just put learns
well bracket jaffa
that's how I'd do it ...

E: Well write down any and all
options that crop up in your mind
'cos there's more than one way to express
that

[writes]
Clause 11
[writes]
- pause -
Clause 12

[grin]
17 Mmmmm
18 because he works hard, jaffa learns well
19 I would guess this is the same as the
20 clauses we've been learning with a
21 semicolon

E: Right

22 If something if something something
[writes]

Clause 13

[writes]
- pause -
23 This one's annoying me because I should
24 know how to do it and I don't

E: Which one is it?

25 jaffa is a good student because ...
26 we did something like this in the lectures
[writes]
27 I'm sure this isn't right, but ...

Clause 14

Clause 15
[writes]

28 ... I'm not too sure about these terms ...
[referring back]
29 of works hard I've not come across anything
30 like this... its an adverb isn't it, well?

E: yeah

31 yeah, and we haven't used that
32 kind of structure before

Subject 1 did not take PART 3
RESPONSES:

Clause 1: Pencil
Clause 2: Sue is female
Clause 3: the cat is called Tom
Clause 4: Mary loves Sue
Clause 5: John loves Henry
Clause 6: Mary is Sue's mother
Clause 7: John is Henry's father
Clause 8: Mary and John are the parents of Sue
Clause 9: Mary and John are Henry's parents

PROTOCOL:

Clause 1

1 Translate that into Prolog?

E: No that's Prolog.

2 That's Prolog?

E: Yah.

Clause 2

Clause 3

3 Are they all nouns?

Clause 4

E: Er no it changes as you go through

Clause 5

4 Right

Clause 6

Clause 7

Clause 8

Clause 9

PART 2

RESPONSES:

Clause 1: ([there, is, a, book]).
Clause 2: ([there, is, a, pencil]).
Clause 3: student(jaffa).
Clause 4: professor(misha).
Clause 5: male(jaffa).
Clause 6: female(misha).
Clause 7: clever(misha).
Clause 8: teaches(misha, jaffa).
Clause 9: admires(jaffa, misha).
Clause 10: learns_well(jaffa).
Clause 11: works_hard(jaffa).
Clause 12: learns_well(X) :- works_hard(X).
Clause 13: good_student(jaffa):-admires(jaffa,misha),
                works_hard(jaffa).
Clause 14: learns_well(X):-teaches(misha,X).
Clause 15: succeed(X):-works_hard(X),
                teaches(misha, jaffa).

PROTOCOL:

Clause 1
   [adds isa later]
Clause 2
   [writes]
   - pause -
Clause 3
   - pause -
       [writes]
Clause 4
   - pause -
       [writes]
Clause 5
Clause 6
Clause 7
Clause 8
Clause 9
Clause 10
   - pause -
Clause 11
   [little scribble]
Clause 12
   - long pause -
       [looks back over other answers]
Clause 13
   [smile, inhale]
   - pause -
       [checks back]
   - long pause-
       [dabs table]
   - pause -
Clause 14
   - pause -
       [write]
   - pause -
Clause 15

PART 3

PROTOCOL:

Clause 1
5 Full stop
Clause 2
6 Er um well it doesn't say that
7 michelangelo hates rafael. It says that two
8 people hate each other

E: Because of ....?

9 The capital letters

E: That's right, great

Clause 3
10 No comma

Clause 4
11 Capital letters

Clause 5
12 No closing bracket

Clause 6
13 No closing square bracket

Clause 7
14 Only one closing bracket

Clause 8
15 Capital B for bach and no full stop

Clause 9
16 Square brackets

Clause 10
17 No minus sign

Clause 11
18 The extra 'knows(Y, Z)' is irrelevant
19 knows(X,Y) is sufficient. Could have
20 had knows(X,Z), heard_of(X,Y)

E: What you're saying is that if you want
to have it like that the arguments are
in the wrong order?

- pause -

   [Nod].

21 Yes
COMPUTERS AND THOUGHT: SUBJECT 3

PART 1

RESPONSES:

Clause 1: pencil
Clause 2: sue is female
Clause 3: tom is a cat
Clause 4: mary loves sue
Clause 5: john loves henry
Clause 6: mary is the mother of sue
Clause 7: john is the father of henry
Clause 8: mary and john are parents of sue
Clause 9: mary and john are parents of henry

PROTOCOL:

Clause 1

1 Ah right, pencil, I'd better
2 write that down.

Clause 2

3 That looks a bit more normal ...
E: if there is anything normal about Prolog ...

Clause 3

4 Ok on this one I'm thinking that it
5 could be that it's a 'tomcat', but
6 actually if it's Prolog it just means
7 that tom is a cat
E: right

8 But ...

Clause 4

9 Oh god we've had so many of those...

Clause 5
Clause 6
Clause 7

10 Seems logical.

Clause 8
Clause 9

11 Which is the logical explanation of
12 all that lot
[laughter]

PART 2
RESPONSES:

Clause 1: there(book).
Clause 2: here(pencil).
Clause 3: student(jaffa).
Clause 4: professor(misha).
Clause 5: male(jaffa).
Clause 6: woman(misha).
Clause 7: clever(misha).
Clause 8: teacher(misha,jaffa).
Clause 9: admires(jaffa,misha).
Clause 10: learns_well(jaffa).
Clause 11: works_hard(jaffa).
Clause 12: learns_well(jaffa):-works_hard(jaffa).
Clause 13: good_student(jaffa):-admisres(jaffa,misha),
   works_hard(jaffa).
Clause 14: learns_well(student,misha).
Clause 15: succeed(student,misha):- works_hard(student).

PROTOCOL:

Clause 1

13 There is a book ... phew ...

Clause 2

14 Here is a pencil ... actually we haven't
15 done any with 'there's' and 'here's' and things.
16 And we haven't used words other than like verbs,
17 I don't think ... likes ... loves ... that's it

Clause 3
Clause 4

18 misha is a professor ... unusual names

Clause 5
Clause 6

19 I like that. A woman's a professor.
20 Feminist Prolog.

Clause 7
Clause 8
Clause 9
Clause 10

E: You're allowed to ask for help ...

21 Well it's like I don't remember coming
22 across adjectives before ... if that's what 'well' is
23 It's an adverb ... so I was thinking
24 I could do learns well jaffa
25 to sort of keep it all into the one
26 word, but I was just wondering
27 if there was something I'd forgotten ...
28 another way of doing it.
29 We've started 'Sir' in the lectures but
   [a teaching exercise]
30 we haven't done anything in the
31 program where it's language ...
32 I don't know if you could do jaffa learns well on
33 the inside of the brackets with a comma
34 between like where in Sir you
35 can say [I, know, that, you, twerp] with commas
36 between each one ... that would
37 be a GUESS ...

E: I think your first guess was better

38 Yup ... 'Cos you'd get very complicated
39 if you went the other way.
40 And I could imagine a list of who learns
41 well and who doesn't learn well
42 if you're trying to think of a list of facts

Clause 11

43 jaffa works hard ... here's another one
   - pause -
   [sigh]
44 it would have to be ...
45 it would have to be the same

Clause 12

46 jaffa learns well ... yuk ... because ...
47 well ... I don't think I've seen a 'because'.
48 I could change that to
49 if he works hard he learns well ...
50 which would be the nearest I've
51 come across I think ... um ... if he works
52 hard he learns well ... he learns well if he
53 works hard ... I'll have to do it in that order ...
54 learns well jaffa if works hard jaffa.
55 Which is the meaning of it anyway.

Clause 13

56 jaffa's a good student because he
57 admires misha so he works hard
58 mmmm ... haven't got good_student ...

E: Is that a problem?

59 It's just different I suppose ...
60 what would I do ... I'd do
61 because he admires ... well I can't do because
62 so I'd do if ... so what the clause would be
63 is he's a good student if admires misha and works hard ...
64 no hard and admires misha ...
65 no the other way round is the best meaning of it ...
admires misha ... if he admires misha ...
good student if admires misha and works hard
or works hard if admires misha works hard if admires misha
good student ... mmmmm, I'll go back to the first one.
But good student seems daft.
I mean we never put an adjective in front of something ...

E: But you solved that one here by just doing that (pointing), so ...

Yeah ...
[repeats what she is writing]
but that isn't the exact meaning of so

Clause 14

All misha's students learn well ...
don't tell me they all admire her ...
al misha's students learn well ...
learn well student misha

Clause 15

Any of misha's students ...
succeed student misha if
it won't work can't put ...
[mutter]
... umm ...
if work hard student ... hmm that'll do.

PART 3:

Clause 1
Well that's ok

Clause 2
[looks back]
Am I allowed to do that.
Yeh that one was right. the
difference is ... yuk ...

E: they don't all relate to one another
so don't worry about that ...

No what I'm thinking is that the
capital letter which would make them not
names but variables, but if you chose to
use a variable like michelangelo

Clause 3
There's no comma in between that

Clause 4
[mutter]
104 Actually I guess that shouldn't be a
105 capital either ...

Clause 5
106 Friends ... the brackets this should be ...
107 you haven't got the end bracket so
108 this wouldn't work ...

Clause 6
109 You haven't got a square bracket in if
110 you're gonna try and do it separately ...

Clause 7
111 [mutter]
111 Parents jum ... um again there is a bracket
112 missing which wouldn't make
113 sense, but it doesn't seem to make much
114 sense ...

Clause 8
115 This shouldn't be a capital

Clause 9
116 No brackets are wrong, it just should be ...
117 there should be round brackets

Clause 10
118 There's um a dash missing

Clause 11
119 [mutter]
120 ... X knows Y if X knows Y
120 ... X knows Y and knows Y Z.

E: Will it work

121 Yes. I would have thought it would have worked
PART 1

RESPONSES:

Clause 1: don't know
Clause 2: Sue is female
Clause 3: Tom is a cat
Clause 4: Mary loves Sue
Clause 5: John loves Henry
Clause 6: Mary is Sue's mother
Clause 7: John is Henry's father
Clause 8: Mary and John are Sue's parents
Clause 9: Mary and John are Henry's parents

PROTOCOL:

Clause 1

1 So what do I do, sorry?

E: That's Prolog and you have to write what the English equivalent of that is ...

Clause 2

2 You want everyday English do you?

E: Yeah

Clause 3
Clause 4
Clause 5

3 Do you want me to put both possibilities for that? ... the inversion or the one that I initially interpreted?

E: No, just put the one ... I know what you mean

Clause 6
Clause 7
Clause 8
- pause -

Clause 9

PART 2

RESPONSES:

Clause 1: don't know
Clause 2: don't know
Clause 3: student(jaffa).
Clause 4: professor(misha).
Clause 5: male(jaffa).
Clause 6: woman(misha).
Clause 7: clever(misha).
Clause 8: teaches(misha, jaffa).
Clause 9: admires(jaffa, misha).
Clause 10: learns_well(jaffa).
Clause 11: works_hard(jaffa).
Clause 12: learns_well(jaffa):-works_hard(jaffa).
Clause 13: good_student(jaffa):-admires(misha), works_hard(jaffa).
Clause 14: learns_well(Students):-teaches(misha, Students).
Clause 15: will_succeed(X):-works_hard(X), teaches(misha, X).

PROTOCOL:

Clause 1

6 Oh, that's like the last lot ...

Clause 2
Clause 3
Clause 4
Clause 5

[pulling face, shaking head]

7 No

[looks back at previous responses to test 1]

Clause 6
Clause 7
Clause 8
Clause 9
Clause 10

8 Hmm. I don't think we've come across this ...
9 jaffa learns well.

E: Right
What's the problem?

10 Could I just do an underscore
11 between there and ... 

[pointing]

E: Yeah

12 Mumble ...

E: That's fine

13 I don't think we've really covered that before

E: Right

Clause 11
Clause 12

[smile]

- pause -

14 Hmm. ... I can rephrase that into an if clause ...
E: Right

15 Is that possible ... or ...

E: It is possible ... um tell me what it is that's thrown you.

16 The 'because'

E: Right ... um ... so do you have any ideas?

17 Hmm, I would do that
    [pointing]
18 and then an 'if' ... ah ... umm colon and dash,
19 and then that ...

E: Right

20 Mmmm ... Yeah
    [nodding]
21 mmm ... you don't think
22 of 'because' and 'if' having the same
23 connotations

E: Right ... would you be able to say why?

24 Mmm.
   - pause -

25 There's something more cer...
26 there's more certainty with
27 because ... you know with an 'if'
    [hand gesture]
28 you know, possibilities, whereas
29 a 'because' is more certain

Clause 13
   - pause -
     [reads it aloud, head shake]
30 No, this so throws me ... it can't be
31 and ... really
   [looks at E for confirmation]

E: mmmm?

32 Because the relations different ... it's a result
33 Hmmm is there a sort of result clause?

E: Not really, no. start writing
what you can put down and we'll take
it from there.

34 Mmmm ...
   [writes]
35 admires misha ...
36 Mmmm
37 And ... comma
E: Yeah

[writes]

38 Mmm

[nodding]

Clause 14

- long pause -

39 Hmmm.

40 To make a clause of 'all misha's students' like

41 student jaffa

[mumble ... mumble ...]

42 will that work?

E: Mmm ... just finish off the clause how

you'd like to ... then we'll have a look

and see whether it would work ...

- pause -

43 ... No

[headshake]

E: It's not going to work

44 No ...

[mutter]

E: OK. We've got ... we know how to do

learns_well jaffa, ok, we've got that ... so

we've got learns_well students. but we don't

know who the students are ... they

could be anybody ... jaffa's going to

be one of them ... there will be others.

But what kind of object do we want to

represent all those students?

45 A variable

E: So we've got this part we can do ...

[pointing]

46 Yeah ummm.

[writing]

47 then the if?

E: Right.

Clause 15

48 Could you have: mishas students will

49 succeed if they work hard

50 and ... um ... student of misha ... um ... hmmm

E: Which is the bit you want to prove?

51 Um ... they will succeed ...
E: Right ... so that's got to be the important clause that goes before the if because the rest is contingent

[nodding, writes mutters]

52 can I put a variable? ...

E: Mmm

53 Anything ...?

[writes]

PART 3

PROTOCOL:

Clause 1
54 No full stop

Clause 2
55 They are both variables, so it's got nothing to work on

Clause 3
56 Comma there

Clause 4
57 That's going to be a variable and it won't make sense

Clause 5
58 No round bracket

Clause 6
59 No square bracket

Clause 7
60 No closing round bracket

Clause 8
61 No full stop

E: Aha true. Anything else?

62 Umm ... well it's going to make it variable

E: Fine

Clause 9
63 No round bracket

Clause 10
64 No hyphen

Clause 11
65 Hmm ... no that isn't needed, that's superfluous ...
PART 1

RESPONSES

Clause 1: pencil
Clause 2: Sue is female
Clause 3: Tom is a cat
Clause 4: Mary loves Sue
Clause 5: John loves Henry
Clause 6: Mary is the mother of Sue
Clause 7: John is the father of Henry
Clause 8: Mary and John are the parents of Sue
Clause 9: Mary and John are the parents of Henry

PROTOCOL:
(No comments by subject).
Clause 1
Clause 2
Clause 3
Clause 4
Clause 5
Clause 6
Clause 7
Clause 8
Clause 9

E: No problem eh?

PART 2

RESPONSES:

Clause 1: book.
Clause 2: pencil.
Clause 3: student(jaffa).
Clause 4: professor(misha).
Clause 5: male(jaffa).
Clause 6: woman(misha).
Clause 7: clever(misha).
Clause 8: teaches(misha,jaffa).
Clause 9: admires(jaffa,misha).
Clause 10: learns(jaffa,well).
Clause 11: works_hard(jaffa).
Clause 12: learns_well(jaffa):-works_hard(jaffa).
Clause 13: good_student(jaffa):-admires(misha),
                              works_hard(jaffa).
Clause 14: learns_well(student):-teach(misha,student).
Clause 15: succeed(student):-teach(misha,student),
                              works_hard(student).

PROTOCOL:

Clause 1

[smile]
1 Well I'll just put book ...

E: That's great ...

Clause 2

[mutter]

Clause 3
Clause 4
Clause 5
Clause 6
Clause 7
Clause 8
Clause 9
Clause 10

2 Hum - I'm stuck on this one I think ... um

E: What's the problem .

3 Well it's gone into an adjective ...
4 you know and usually its just two nouns
5 and a verb ... would it be learns
6 jaffa well? ... yes?

E: There are several ways of doing it ... that's one of them.

Clause 11

7 Mmm ... I've just thought of another way
8 I think, ... um ... you could put works hard ...

Clause 12

9 Is there a special symbol for 'because'?

E: No

10 Oh ... right ...
   - pause -
     [laugh]

E: What were you thinking of doing?

11 Well I was thinking of having
12 another set of brackets ... ah ... I don't
13 know really.

E: What kind of Prolog clause would it be?

14 Um ... one with a conjunction,
15 I can't remember whether they have a special
16 name or not ...

E: And what do you mean by a conjunction

17 Well one with a plus ... you know where
18 there are two things in a line ...

E: Do have any ideas yet?

19 ... well ... um ... well just if I put
20 learns jaffa well and then because and
21 that bit in brackets? the next one he works hard?

E: Show me ...

[writes]
22 I want to put and he works
23 hard, but because ... if I just put
24 because it's going to be another ... it's
25 not going to relate to the two

E: OK. We don't have a Prolog sign for
because so we can't use that.

26 But there's one for 'if', is that right?

E: Yes

27 If he works hard then he learns well.
28 I didn't need to cross that out
29 after all!
30 Do you have to put jaffa again, or can
31 you use he, you need to put jaffa ...

E: Yeah

Clause 13

32 ... Hmm ...
- pause -
33 ... if he works
34 hard he is a good student and we've
35 got to fit in the admiring bit ...
36 could you have two sort of ifs, or can
37 you only have one?

E: You can only have one 'if'

38 Hmmm

E: But you can join things up in other ways ...

39 Yeah ... um

E:

[pointing]
40 So you already ...
41 Yes it's ok, I've just sort of started,
42 just writing things in ...
- pause -
43 um ...
- pause -
44 ... I could that he admires
45 misha and he works hard, but I
46 suppose you could have that but it seems
47 a bit strange to say that he's
48 a good student because he admires misha

Clause 14

49 I've been thinking about my program,
50 you know my mind's on that, you know
51 every something is a something ... um ...
[writes]

Clause 15
[goes back to Clause 14]

52 Oh I'll cross that out
[crosses the s off students]

E: Right

PART 3

PROTOCOL:

Clause 1
53 There's no full stop at the end

Clause 2
54 Capital letters

Clause 3
55 No comma

Clause 4
56 That can't be a variable

Clause 5
57 There's no closing bracket

Clause 6
58 There's no square bracket

Clause 7
59 That's ... can't be a variable. There's no closing bracket

Clause 8
60 There's no full stop ... oh and also
[points to argument]

Clause 9
61 They should be round brackets

Clause 10
62 What's that?
[laugh]
Yes there should be a

Clause 11

E: Right. Would that rule work?

Um ... it says that X has heard of Y. I suppose you don't need the one with X Z, you don't really need that.
PART 1

RESPONSES:

Clause 1: don't know
Clause 2: Sue is female
Clause 3: Tom is a cat
Clause 4: Mary loves Sue
Clause 5: John loves Henry
Clause 6: Mary is the mother of Sue
Clause 7: John is the father of Henry
Clause 8: Mary and John are the parents of Sue
Clause 9: Mary and John are the parents of Henry

PROTOCOL:

Clause 1
Clause 2

1 Sue is female

Clause 3

2 Tom is a cat

Clause 4

3 Mary loves Sue

Clause 5

4 John loves Henry

Clause 6

5 Mary is the mother of Sue

Clause 7

6 John is the father of Henry

Clause 8

7 Mary and John are the parents of Sue

Clause 9

8 Mary and John are the parents of Henry

PART 2

RESPONSES:

Clause 1: there_is_a(book).
Clause 2: here_is_a(pencil).
Clause 3: student(jaffa).
Clause 4: professor(misha).
Clause 5: male(jaffa).
Clause 6: woman(misha).
Clause 7: clever(misha).
Clause 8: teaches(misha, jaffa).
Clause 9: admires(jaffa, misha).
Clause 10: learns well(jaffa).
Clause 11: works hard(jaffa).
Clause 12: learns well(jaffa):-works hard(jaffa).
Clause 13: good_student(jaffa):-admires(jaffa, misha),
          works hard(jaffa).
Clause 14: learns well(misha, students).
          [Then writes]
          learns well(student):-teaches(misha, students).
Clause 15: will_succeed(X):-work hard(X),
          student(X, misha).

PROTOCOL:

Clause 1

9 Well I can do a isa,
   [writes is_a]

10 Then there's got to be a bracket. but
11 you can't put things like that inside
12 brackets, whatever you call them
   [points to the 'there']
13 Can you do a there_is_a?

E: You could do that ... 

14 If I linked them all together with
15 underlines

E: You could ...

16 'cos ... er ... well it would make human
17 sense ... I mean it would just be book
18 there is a ...

E: Right

19 Well I'll do that because ... er ...
20 because I can't see if there's an
21 alternative ...

Clause 2

22 Another positional one

Clause 3
Clause 4
Clause 5
Clause 6
Clause 7
Clause 8
Clause 9
Clause 10
- pause -
Clause 11
Clause 12
- pause -
  [laughter - writes it straight down]
Clause 13

23 I'm not sure if this is quite right,
24 but er ... that's the only way I can
25 think of putting it ...

E: That's fine. What are you worried about?

26 Mainly because it's um ... um ...
27 because he admires misha so he er works hard,
28 which isn't quite the same as and he works hard ...
29 he works hard because he's
30 admiring misha ... not and he works hard, so
31 ... I don't think that makes
32 quite the same English sense ...

E: Right, ok.

Clause 14

  [writes a little. Stops]
  - long pause -
  [giggle]

33 Erm ...

E: What's hard about that one?

34 All misha's students learn well ...
  - pause -
35 um ... if I did it as I've done
36 here it would be um learn well misha's
37 students ... no ... it would be misha
38 learns well ... teaches students ... yes,
39 yes ... I'm not sure ... if you could
40 put students in that ... so ... that says misha
41 learns well students ... which
42 doesn't make much sense
  [laugh]
43 um ... could I change that to teaches well,
44 misha teaches well students ... but even so
45 I don't know whether that's valid ...

E: Ok

46 You'll have to accept that I'm not
47 very sure on that
Clause 15
- pause -

[smile]

PART 3

PROTOCOL:

Clause 1

[mutter]
48 There's no full stop

Clause 2
49 Ahm capital letters

Clause 3
50 No comma

Clause 4
51 Capital letters

Clause 5
52 No rounded brackets

Clause 6
53 No square bracket

Clause 7
54 Mmm ... missing another rounded bracket

Clause 8
55 Capital b ... ah and no full stop as well

Clause 9
56 Uhm ... you need rounded brackets somewhere

Clause 10
57 Aahm ... you missed out the dash

Clause 11
E: What do you make of that?

58 X has heard of Y if X knows Y and
59 Y knows Z ...

E: Will it work?

60 I can't see any reason why not.
Computers and Thought: Subject 7

Part 1

Responses:

Clause 1: don't know
Clause 2: Sue is a female
Clause 3: tom is a cat
Clause 4: Mary loves sue
Clause 5: john loves henry
Clause 6: mary is sue's mother
Clause 7: John is henry's father
Clause 8: Mary and John are sue's parents
Clause 9: Mary and John are henry's parents

Protocol:
Clause 1
Clause 2
Clause 3
Clause 4
Clause 5
Clause 6
Clause 7

1 When I put Mary is Sue's mother and
2 John is Henry's father, it could be the
3 other way round, shall I put that in too?

E: No. You've said it -
I know now that you know, so it's ok.

Clause 8

4 Yeah. This is a similar sort of thing isn't it?

E: Mmmm

Clause 9

5 Actually I never quite got to grips
6 with these three things in brackets

E: Mmmm ... in what way?

7 Well I never quite understood why, um,
8 you can know it's Mary and John who
9 are the parents of Henry and yet you
10 haven't said specifically ... you know I
11 don't understand that the computer knows
12 that that's what you meant ...

E: Right

13 Rather than that Mary is a parent
14 of John and Henry ... 'cos it doesn't
E: Yes

15 Y'know think that necessarily ...

E: Right, well the only way that Prolog knows - it doesn't know actually - the only way it can figure anything out is because your consistent about what you're doing ...

16 Mmm

E: So when it will become critical is when you ask a question, and when you ask the question X and Y , say if you, you decided when you set up the database that Mary and John were going to be the parents of Henry ok? that's just your decision so that's .... if you started asking the questions parents and then putting Mary X Y, then you will still get John and Henry ...

17 Yeah, yeah, it's consistent ...

E: So it would be silly to put it as a fact one way, and ask a question the other way, 'cos you'll get a daft answer ...

18 Mmm [nodding]

PART 2

RESPONSES:

Clause 1: is(book).
Clause 2: here(pencil).
Clause 3: student(jaffa).
Clause 4: professor(misha).
Clause 5: (omitted)
Clause 6: (omitted)
Clause 7: clever(misha).
Clause 8: teaches(misha,jaffa).
Clause 9: admires(jaffa,misha).
Clause 10: learnsWell(jaffa).
Clause 11: workshard(jaffa).
Clause 12: learnsWell(jaffa) :- workshard(jaffa).
Clause 13: goodStudent(jaffa) :- admires(jaffa,misha), workshard(jaffa).
Clause 14: learnsWell(X) :- teaches(misha,X).
Clause 15: willSucceed(X) :- teaches(misha,X), workshard(X).

PROTOCOL:

Clause 1

19 If that's just a basic assertion well
20 you just put a ... what was in that
21 one with a pencil
[reaches for card]

22 you could just put a full stop in?

E: That's right. That's what that one did.

Clause 2
- pause -

23 will that do?

E: Yep

Clause 3
Clause 4
Clause 5
Clause 6
Clause 7
Clause 8
Clause 9

24 I also could have put those the other way - those two

E: Right

26 It makes more ... you remember better

Clause 10
Clause 11
Clause 12
[starts writing]

27 Because - that's not the same thing as 'if' is it?

E: Not to us, no

28 Is it to them?
[laughter]

E: Well what do you think?

29 Yes

E: Right - 'cos we don't have a symbol for because ... in Prolog

30 We don't? I thought for a moment
31 perhaps we did

E: No

Clause 13

[laugh]
32 Have we got a good student?
   [looks back, then writes]

Clause 14
   - pause -
   [laugh]
33 Um ...
   [scrutinizes paper]
E: What are you thinking of doing?
34 Well getting in ... um ... I don't know
E: What's made you stop writing?
35 Um well I've got to be able to get
36 out something that's already here ... um
E: Well you've got learns_well
37 I've got learns_well
E: Right ... and before you had jaffa, but ...
38 Oh yes I remember, learns_well.
39 someone learns_well because they are students ...
E: Of misha
   [mutter]
40 X learns_well if X
41 Is a student of ... misha

E: Great

Clause 15

42 Mmmm. Oh dear.
   [does it quickly]

PART 3

PROTOCOL:

Clause 1
43 Oh can it just be little things like the full stop
E: Yep

Clause 2
44 Capitals

Clause 3
45 No comma in the arguments

Clause 4
46 Capital letter
Clause 5
47 No closing round ...

Clause 6
48 No closing square ...

Clause 7
49 You've got a funny bracket in there!

Clause 8
50 Capital b ... oh and a full stop

Clause 9
51 Um ... well you want round brackets ...

Clause 10
52 No ... umm ...
    [signals the missing hyphen]

Clause 11

E: Do you understand what that rule does, and does it work?

53 X has heard of Y if X knows Y and y
54 knows Z. Well you haven't got a Z this side
PART 1

RESPONSES:

Clause 1: don't know
Clause 2: Sue is female
Clause 3: Tom is a cat
Clause 4: Mary loves Sue
Clause 5: John loves Henry
Clause 6: Mary is Sue's mother
Clause 7: John is Henry's father
Clause 8: Mary and John are Sue's parents
Clause 9: Mary and John are Henry's parents

PROTOCOL
(Subject made no comments).
Clause 1
Clause 2
Clause 3
Clause 4
Clause 5
Clause 6
Clause 7
Clause 8
Clause 9

PART 2

RESPONSES:

Clause 1: is_a(there,book).
Clause 2: is_a(here,pencil).
Clause 3: student(jaffa).
Clause 4: professor(misha).
Clause 5: (omitted)
Clause 6: (omitted)
Clause 7: (omitted)
Clause 8: teaches(misha,jaffa).
Clause 9: admires(jaffa,misha).
Clause 10: learns(jaffa,well).
Clause 11: works(jaffa,hard).
Clause 12: learns(jaffa,well):-works(jaffa,hard).
Clause 13: is_a(jaffa,goodstudent):-admires(jaffa,misha),
  works(jaffa,hard).
Clause 14: learns(X,well):-student(X,misha),
  work(X,hard).

PROTOCOL:

Clause 1
Clause 2
- pause -
Clause 3
1 Ha ha. Hmmm ... I'm not very clear
   about my Prolog clauses anyway

Clause 4
Clause 5
- pause -
Clause 6
Clause 7
Clause 8
Clause 9
Clause 10
Clause 11

- pause, smile, cough -

E: What's the problem?

- pause -

3 hmmm ...

- writes -

Clause 12
- pause -
Clause 13
Clause 14

4 Can I skip?

E: um

5 No

E: Nope

   [laughter]

Tell me about it ...
tell ... what's hard about it, which
particular bit's difficult?

6 All misha's students learn well ...
7 I'm not sure how to put it into Prolog
8 clauses

E: mmmm

9 Unless I use the ... er ... I can't even
10 remember what the ... I think this
    [pointing]
11 you have to put in a general rule.

E: right ... so

12 If ... X is misha's student then X
13 learns well

   [nods writes].
E: You're better than you thought you were

14 Well ...
   [chuckle]

Clause 15
- pause -

15 hrrrrgh!
   [laughs]
16 Any of misha's students who work hard
   [repeats]
- pause -
   [writes]

E: That's right - super

18 I think I've got something wrong here

PART 3

PROTOCOL:

Clause 1
19 Full stop missing

Clause 2
20 Capital M and S ...

Clause 3
21 Comma missing

Clause 4
22 Capital ...
   [points]

Clause 5
23 Bracket missing

Clause 6
24 Square bracket missing

Clause 7
25 Another round bracket missing

Clause 8
26 Full stop missing ... and the capital

Clause 9
27 Er ... 2 round brackets

Clause 10
28 The hypen's missing

Clause 11
E: Now - what does that mean to you and will it work?

29 Oh. X has heard of Y if X knows  
30 Y and Y knows Z. this rule will work for  
31 this rule but not this rule,  
32 because Z is not instantiated.
COMPUTERS AND THOUGHT: SUBJECT 9

PART 1

RESPONSES:

Clause 1: don't know
Clause 2: sue is female
Clause 3: tom is a cat
Clause 4: mary loves sue
Clause 5: john loves henry
Clause 6: mary is sue's mother
Clause 7: john is henry's father
Clause 8: mary and john are sue's parents
Clause 9: mary and john are henry's parents

PROTOCOL:
(Subject made no comments).
Clause 1
Clause 2
Clause 3
Clause 4
Clause 5
Clause 6
Clause 7
Clause 8
Clause 9

PART 2

RESPONSES:

Clause 1: don't know
Clause 2: don't know
Clause 3: student(jaffa).
Clause 4: professor(misha).
Clause 5: (omitted)
Clause 6: (omitted)
Clause 7: (omitted)
Clause 8: teaches(misha,jaffa).
Clause 9: admires(jaffa,misha).
Clause 10: learns_well(jaffa).
Clause 11: works_hard(jaffa).
Clause 12: works_hard(jaffa):-learns_well(jaffa).
Clause 13: good_student(jaffa):-admires(misha), works_hard(jaffa).
Clause 14: succeed(X):- teaches(misha,X),learns_well(X).
Clause 15: succeed(X):- teaches(misha,X),works_hard(X).

PROTOCOL:
Clause 1
Clause 2
Clause 3
Clause 4
Clause 8
Clause 9
Clause 10
Clause 11
Clause 12
1 Mmm. These are difficult

E: Right.
What's hard about it?

2 I know 'and', and 'if'
3 but ...
4 Um ...
5 Could you use if?

[nods]
[writes]

E: Ok, so read out to me
what you've written in Prolog.

6 Er, jaffa works hard if jaffa learns well

E: Right, and ...
[laughter]
what do you think the English means?

7 Um. The English of this,
8 what I've written?

E: No, this English, sorry
[pointing]

9 jaffa learns well because he works hard ... um
10 Yeah ...
11 Should be the other way round ...

E: Mmm
Do you think?

12 Um ... jaffa learns well if jaffa works ...
13 Yeah ... it might make more sense

E: Wait a minute
Let's have a think.
jaffa works hard if he learns well, ...
Um ... jaffa learns well if he works hard ...
Which do you think yourself?

14 Mmm ... Yes.
15 The other way.
16 jaffa learns well if he works hard.

E: Right.
Are you happy about substituting 'if' for 'because'?

17 Um ... it doesn't really mean the same does it?

E: No

[writes]
- pauses -

18 And so he works hard ...
    [writes looks at experimenter]
19 Is that right?

E:    [nodding]
Yeah, that's fine.

Clause 14

20 Um ...
    [writes]
E: Right
    [nodding]

Clause 15

21 Um put that ... um
22 in there ...
23 That's up there ...

E: Right. Yes, yeah ...

PART 3

PROTOCOL:

Clause 1
24 No full stop

Clause 2
25 Capital letter

Clause 3
26 No comma

Clause 4
27 There's a capital in there

Clause 5
28 Um no round bracket

Clause 6
29 No square bracket

Clause 7
30 Just one bracket

Clause 8
31 Capital letter

Clause 9
32 Oh, square brackets, should be round

Clause 10
33 There's no dash

Clause 11
E: Now - is that rule ok and will it work?

34 Um ...
35 No it should be X,Z.
APPENDIX 4

Protocol Transcript for Julie and Mary

Note:

The protocol contained a great many repetitive phrases. Since these did not appear to contribute anything to the protocol, they have been taken for easier reading.

QUESTION:

Can you give a mutually satisfactory explanation of what this clause does, and how it does it?

foobaz(X, [X|_]).
foobaz(X, [_|Y]):- foobaz(X,Y).

JULIE: oh, tricky eh?

1 MARY: do you know what that is?

2 JULIE: it's to do with the head of the list it's that...
3 it's to do with lists and this is
4 this system for taking this
5 first part of the
6 list

7 MARY: mmm

8 JULIE: isn't it
9 and then for moving down the list

10 MARY: I know it's connected to something
11 there though and there....
14 X is going to be in the list
15 Y is going to be in the list (mumble)

16 JULIE: it must be the thing for splitting up the list
18 mustn't it it's got to be.
19 That's they need the two you need
20 I'm sh...
21 uuh God, yes, I don't know
22 You need the two clauses you see don't you
23 when you're splitting up the list
24 because they've got to be able to
25 take the head of the list
26 I think (chuckle)
27 and then take the rest of the list
28 right
29 are you with me?

30 MARY: mmm
31 JULIE: so you do the fi...
32 I think you have a clause first of all
33 to be able to take the first thing in the list
34 and then to take the second thing in the list
35 you go, you have another clause ..... 

36 MARY: mm

37 JULIE: to show you how to do that
38 or another rule

39 MARY: mm

40 JULIE: to show you how to do that

41 MARY: & JULIE: together (SOMETHING)

42 MARY: I would have said actually
43 that was the stopping clause
44 so that's.... cos this
45 that is a recursive rule I think
46 'cos that's on the left and that's
47 on the right
48 [tapping]
49 so when it gets to foobaz when
50 it's got X and it's got X in the list
51 that's it it stops
52 [taps]
53 although I don't know why
54 [smile]
55 cos that
56 I've not come across that blank
57 [indic. underscore variable]
58 before or the underscore

56 JULIE: no neither have I
57 what do you think it means then?
58 I'm just assuming it's the normal thing
59 that we've heard
60 I mean that it's just a space
61 [looks at Mary - who does nothing]

62 Oh I see what you mean
63 no we don't know what that means do we?

64 MARY: if it does mean something
65 or if it's just saying
66 that could be anything

68 JULIE: so you think..
69 [interrupt]

70 MARY: I would expect another variable there
71 a list or a trail at the end here
JULIE: and it hasn't got anything

MARY: no

JULIE: so you're saying that this is a recursive thing

MARY: mm

JULIE: to go through a list

MARY: mm

and it's going to keep going round like this

[circles pencil]

doesn't want to stop it because when you've got X here

and you've got X in the list

it's going to stop.

this is confusing me because

[indic. underscore variable] I don't know what it means

but can't, I mean,
doesn't that explain to you then what

what that means

I mean I just don't understand why

it's recursive if that if that

the second bit the foobaz X Y if

[indic. the recursive call]
on the... why do we need a second bit on this?

if that's recursive?

what do you mean the second bit?

why do we need this if clause here?
in that form ...
do you see what I mean?

I don't understand why we need it in that form

because I mean that, presumably, right

I'm going to take back what I said.

I'm going to say actually I don't thing it is recursive

I don't think it is recursive

no, no

but I....

If you know, if you think it is...

Now why don't you think it's recursive?

because it would have some other

something else down here

it would have um something else possibly
121 JULIE: so do you think it is just simply a list?
122 it's talking about lists or not?
123 MARY: I don't understand why it's set up like that
124 Its going to make up a list of foobaz isn't it?
127 JULIE: yup
128 MARY: and lots of ... all the foobazs that it knows
130 JULIE: yep
131 MARY: could fit into the list
132 JULIE: yah
134 MARY: so it's going to start
135 it's going to be given
136 a list to make up of foobaz X Y Z
139 do you think? or...
142 JULIE: Well what I don't understand is
143 what I'm not quite sure what this format is
144 do you see what I mean
145 MARY: this?
146 JULIE: yeah
148 MARY: that is a list
149 JULIE: yeah
150 MARY: that..you know what that means?
152 JULIE: yes I mean that that's the bit I understood initia...
154 straightaway but I don't really understand the complete
156 set-up of having a variable and then...
158 MARY: that's the start...
159 that will always be the start of the list
162 um ...I don't know why it's a blank though
164 JULIE: you see if says it's like
165 it's like that ok ... it's foobaz X Y
168 MARY: mmm
169 JULIE: I mean [looking at E] am I getting ridiculous
171 or is that ... does that mean that that should be X Y?
173 does that mean that's what we are looking for there ...
175 [interrupt]
177 MARY: but it needs to be more than
177 X Y because we need a list
made up with X and Y in it
but it might have something in between X and Y.
Y is going to be at the end
of the list I think
and you can have anything between X and Y.

JULIE: yeah but that variable is going
to be instantiated isn't it?
I mean those variables

[interrupt]
MARY: X Y will always be instantiated

JULIE: yeah

MARY: um Y is going to be pushed to
to the end of the list all the time

JULIE: but is that the only one that will fit there,
that oh I see what you mean...
yeah I see what you mean .. wait a minute I think
I'm beginning to understand what you mean.
Well what's this X at the beginning here then?

MARY: I think that's the start of a list

JULIE: yeah but I just don't understand why
it's in that format then.
I don't understand this

MARY: um .. if you want a list with this X at the front
and you want to get from X to Y ....
list with X and Y in it

MARY: so you're going to scan the list of Xs and Ys
it's not likely that X and Y will be in...
will match straightaway um otherwise
there wouldn't be any point in having that list
it's going to instantiate X to start with
and then it's going to go down and find a Y
and Y is going to be pushed ....

[interrupt]
JULIE: I think I do

MARY: into the list

JULIE: Yeah I think I see what you're trying to say
But what I don't understand is why
if that's instantia... I mean that...
oh wait a minute wait a minute.
do I see what you mean
or do I not see what you mean?
MARY: [to E] I want to ask something

E: Yep

MARY: ...this is confusing me
this underscore here

E: Ok

MARY: with nothing in it

E: That underscore means that um
you don't care what else there is ..
it's the same thing as a variable name
but it means
don't ever bother trying to find out
what the value of this variable is
because I don't care

MARY: but that's what you're trying
to find out
by using this rule aren't you
you know what X and Y is
and you want to find whatever
comes between X and Y

E: Uh
well just do another re-read
knowing now that that is a variable name,

that underscore

E: yeah

B: Ok

MARY: mmm

E: In what it might contain

MARY: Hmm

E: Um

remembering that the und...is
a variable, like X
but also the underscore
can match an object and a list, either

MARY: Uh uh

JULIE: I mean my instant reaction right would be
to say that we are
it's looking for two halves of a list.
Or I mean it's looking for the thing that's
at the beginning of the list
and then it's looking for the
rest of the stuff that's in the list

MARY: I wouldn't have thought...

JULIE: no?

MARY: that it's looking for the beginning
because you give it the beginning
by giving the X.
Its looking for what's in between X and Y
that's .... that's how I see it.
We know what the beginning is,
it's X presumably,
and we know what the end is, Y
and it's going to find out what's in between.

JULIE: No, say you got this in your program, right,

MARY: mmm

JULIE: that's in there,
what sort of questions are you going to ask that
that's the only way I can think of it

MARY: um

JULIE: so you're gonna ...
if you ask it a question

MARY: mm

JULIE: like that

MARY: I would say if you're going
to ask it a question
you'd be asking it um
foobaz X Z

JULIE: Yeah but what do you mean by that?
I mean why would you be aski... yes
so why ... and then what would it do?

MARY: um
[grins]

JULIE: yes that's it you see
cos that's what I don't under..
I mean the way you're talking
I don't really understand what the..

[interrupt]

MARY: I keep wanting to picture in my mind
something like I've already done um
um [draws] X Y you know these chains
[writes X Y as in a ]
[transitivity relation]
345 JULIE: yes, quite, yeah

348 MARY: and I keep thinking that
349 foobaz X and Y would be here
350 foobaz Z would be here
352 and you'd want to get from X to Z using that

354 JULIE: oh you think that's the er
355 you see I don't even .. I don't even
357 I can't sort of visualise
358 that thing you're saying it is then...
360 Yeah I'm just visualising
361 that in the square brackets
362 as if it's just a list, y' see

363 MARY: well it could be quite a
364 few in between here
   [indicates drawing]

365 JULIE: exactly

366 MARY: yeah

367 JULIE: yeah quite .. yeah I see that
370 but you see what I don't...
371 That first clause
372 X comma, right
373 I think I ..I think
375 I think I understand what that means,
376 that you want ...you want to get what's at the
378 start of the list don't you
379 so do we agree on that
380 that that's to find out...

381 MARY: mmm

382 JULIE: what's at the start of the list? Or...

384 MARY: I s... yeah
   [taps]
386 Yeah I would agree

387 JULIE: so we agree
388 so this bit then is to find out...
390 now all right
391 so if I've got that bit without the foobaz X Y
393 I would think oh it has to find
394 out what's the rest of the list, yeah?

396 MARY: mmmmmm ...if there is one.
400 You might already know that
401 er we might already have a
402 foobaz X X in which case it
403 wouldn't go any further

405 JULIE: no exactly
406 ah well that's what you're
talking about isn't it? that's it.
That means that it's got to be
something different

MARY: mmmmm

MARY: yeah

JULIE: is that it? I think
so in that sense it probably is recursive then?

MARY: that's what I thought

JULIE: yeah? or not?

MARY: to start with I wondered whether you
perhaps wouldn't have X and Y in this list
and you'd want to find X Y given that you already
know X Z, and X Z is...

JULIE: oh yeah
I was just trying to wipe everything away then
and just think of it cold just like that
and why ... what the purpose of that is
and the purpose of that would seem to me
is to m ...is to make sure that X and Y...
in other words the two components of this thing
aren't the same but don't represe...

JULIE: I'm not sure if I ever was ... um

JULIE: I'm sure it's parts
I'm sure it's parts of a list
it's looking for anyway
whether it chooses numbers

MARY: no

JULIE: or not
it doesn't matter does it?
but I mean they want the start
you want the start of it
and then you want the rest of it

MARY: mmm ... yeah

JULIE: and then you want
but you want them so they're not the same thing

MARY: mmm
JULIE: so that the two parts of the list aren't the same thing.

MARY: what do you mean the two parts of the list aren't the same thing? I see this as ...

JULIE: go on

MARY: being the start of the list

JULIE: yeah

MARY: and this is the end of the list and you want to know

[interrupt]

JULIE: No, you see, no, what I'm thinking I thought that this X here was that... that is the number we're looking for it's the thing we're looking for isn't it that's only there - the X comma is only there to say this is what we're looking for out of this so then you have the list. That's what I think it means then X is what you're looking for out of this list. So here you're looking for the X, X which is the beginning of the list here you're looking for ... well I mean the beginning.

MARY: You'd have to have something

JULIE: of the list which is, no?

MARY: You'd have to have something to start off the list though, and I would say X was the start

JULIE: yeah

MARY: it's the list you don't know it's this bit you don't know [indic. underscore characters] this bit you're going to build

JULIE: but that's the same there isn't it?

MARY: why

JULIE: it's that bit you don't know it's the second part you don't know in the first bit isn't it
as well

JULIE: no? Are we being confused by this underscore do you think

[smiles]

MARY: yes

JULIE: I think we are aren't we? 'cos I'm sure we'd be able to do it if we had ordinary variables there

MARY: it could be a list well it is a list isn't it

JULIE: well yeah ... so yeah it doesn't matter how many things it's got does it, that's it.. yeah, yeah

MARY: don't you think this means that we know what X is we know what Y is X is the beginning of the list Y is the end of the list what we're looking for is this inbetween the list

JULIE: well no I don't really but I think there's something really basic here that we're miss.. I mean I think this is really obvious and we're com... we must be complicating it I'm quite sure. What's the purpose of this is gonna be this? if this is... I don't understand what the [mumble] is I don't understand what the purpose of this if' clause is ...I mean it's got to have there must be a reason mustn't there

MARY: well um

JULIE: but I don't know what it is I don't understand why

MARY: it's going to go looking through,through these assertions for X and Y actually it's more likely to have Z Y [writing] so that um it's got X all the time here

JULIE: yeah

MARY: it'll go up and down the list first of all it'll come to Z
397

578 so it'll be X and Z
579 JULIE: yeah

580 MARY: and Z will get put in the list
581 JULIE: yeh

582 MARY: then it'll go back up and it'll have X again
583 no it'll have Z at the beginning of the list
584 here as well as here
585 JULIE: yeah

586 MARY: um I think all the time
588 it remembers what X is though
589 so it's looking for something that matches...
591 I'm getting confused now

592 JULIE: so you're say...

593 MARY: it's matching

594 JULIE: you're saying it is recursive then
595 is what you're saying

596 MARY: I think it is

597 JULIE: yeah

598 MARY: and it keeps snatching things out of here
599 and keeps pushing them along
600 to get from X to Y

601 JULIE: presumably these two
602 these two things
603 go together don't they though
604 so how does this one react with this one then?

605 MARY: I think that that
606 if it is recursive

607 JULIE: Right, yeah

608 MARY: I think that is the stopping clause
609 so that when it's got X and X
610 it's not going to go any further
611 it's got what it wants

612 JULIE: well it's been through the list and it...
614 oh I see what you
615 I think I see what you mean.
617 So tell me what your explanation is again then
618 just so as I understand
619 cos I don't think I agree
620 but I can't
621 come up with anything that's more sensible
MARY: um
[writes]
[long pause]

JULIE: but you see your

MARY: I don't think it would need to know X Y because it would already ....

JULIE: well we're presuming that X...
we're gonna have facts presumably in there aren't we?

MARY: yeah
yeah I think you would want to know X Y you wouldn't have X Y in the list of facts it would be something like this so you'd want to get from here to here like those um what was it every animal..

JULIE: I see the logic of what you're saying I just it's funny I just don't understand how it actually does it then and why this is here y'know I think I'm being thick or something

MARY: that that is telling it it's going to look in these assertions that it already knows to see if it can find X Y

JULIE: yah

MARY: so each time it's going to go down and it'll say well have we got X X, no have I got X and then something else and it'll put that something else in the list

JULIE: All I was I ... You see I don't I don't understand... To me this is the list isn't it?

MARY: mmm

JULIE: oh hang on a minute

MARY: it doesn't have to be a list (if you're already there (JULIE: oh, sorry, sorry, (sorry, yeah JULIE: sorry I see it now I think I'm looking at it completely the wrong way yeah, sorry, right
399 yeah ok ... so the squ...
679 yeah, oh, yeah so ...yeah sorry
683 so these round brackets
684 are these brackets
685 this is X Y then really.
687 I mean that's why I didn't understand
688 the relevance of that I think,
690 cos this is a list within this
691 within the brackets isn't it
692 after the X

693 MARY: mmm

694 JULIE: so what you're saying
695 you're saying this is ...this
698 this X here is equivalent to this here
699 doesn't mean it's the same th..
700 well it would be the same thing
701 in this case
702 is that what you're saying
703 that this X here is the first part of that clause

705 MARY: mmmm

706 JULIE: fine, ok ....so
708 what I don't understand then then
    [laugh]
709 is how it works
710 now I've got I think I've got that
711 now I understand that bit of it
712 so what does this bit actually do
713 what's this bit doing

714 MARY: which one?

715 JULIE: these these bits in the square brackets, sorry

717 MARY: it's finding a route from X to Y
718 it's finding the list

719 Together: between X and Y

720 MARY: if there is one

721 JULIE: it's working down a list you mean yeah?

723 MARY: it's working down here, up and down
APPENDIX 5

Protocol Transcripts for MSc Students
Backtracking Exercises

Note:

The Appendix is organised by Subject rather than by task. Not all Subject did the version containing the 'cut'.

BACKTRACKING EXERCISE: SUBJECT 1

1 Well f will instantiate X to be 3
2 so b will be instantiated to be 3
3 I can't see that any of this will work at all
4 because they're all instantia ...
5 I mean e's instantiated ...
6 If e's ... if e's instantiated to
7 if X is instantiated to 1 here
8 then d would have to be 1
9 and b would have to be 1
10 and they'd all have to be the same number
11 all the way across,
12 wouldn't they, for X,
13 for any of them to work?
14 So I can't see that
15 any of them would come out with X being anything
16 because they are all
17 instantiating X to be something different.
18 Oh except b would ...

[pause]

E: Well start from clause 1

19 Right

E: So a of X if b of X

20 Yeah and c of X ...
21 but you would have to be satisfied by either of these
22 in which case it would be d and e being the same one

E: Ok so you've got b now

23 That would be 1,
24 so you've got 1
25 so you have to get a c being 1,
26 but you've got no c being 1
27 you've only got it being 3,
28 so that first one would fail.
29 The second one, b, would have to be instantiated
30 The only way you can get b instantiated is from this,
31 so it would be 3,
32 in which case d would have to be 3
33 and you haven't got a d being 3,
34 you've got no e being 3,
35 and that one would fail
36 in which case X could only be 3 ...
37 Sorry b could only be ...
38 It's only this clause which would succeed
39 because that's the only one where two numbers
40 don't have to be two variables ...
41 The \(\chi\) isn't repeated twice
42 so you don't need the ah ...
43 so this is the only one that wouldn't fail
44 so you'd get b of X equal to 3 ... right.
402

MSC. COMBS AND STELL BACKTRACKING EXERCISE: SUBJECT 2

1 From a call of a(X)?
2 Well the Prolog system would look for the first premiss
3 as it were in the argument
4 which is b(X)
5 and then it would call this procedure b(X)
6 which would look then at the premisses of that argument
7 which are d(X) and e(x)
8 and would call the first
9 which is d(X) ... um
10 and then it would instantiate X in d(X) to 1,
11 no, because 1 is the first one there
12 um and then it would look at e(X)
13 and it would instantiate X again to 1
14 because e is the first one there
15 and that's a 1.
16 um hmm.
17 I'm not quite sure then what that
18 X would be instantiated to
19 because there's not,
20 let me see that would be a 1,
21 and that would be a 1 so um,
22 um so that is saying b(X)
23 if d(X) and e(X)
24 and it finds those
25 instantiated to 1,
26 so I take it that X would be instantiated to 1.
27 I'm not terribly sure about that actually,
28 but its next call at any rate would be
29 would go back to there
30 and it would find another call of b(X)
31 which is here
32 and it would be able to call that
33 if there was f(X),
34 and it would find f was 3
35 um ... oh actually it might not do that
36 because if it's already instantiated that in fact
37 it wouldn't go on to do that
38 ... so then it would call ...
39 because it's already been able to do b(X)
40 it would only call second b(X) if it
41 couldn't do that particular one,
42 so then it would go to call on c(X)
43 and there is no procedure c
44 so it would go straight to the database
45 and find c was 3
46 um and so it would instantiate that X to 3
47 um and it would succeed
48 but I'm not quite sure what that X
49 would be instantiated to
50 because you have a 1 there and a 3 there,
51 and there's no um ...
52 there's nothing to say they should be
53 added together or anything ...
54 So I'm not quite sure what that X
55 would be instantiated to,
presumably to the first one which would be a 1,
but um I'm not sure about that.
But as far as I can see
that's the way it would work.

E: Right ... if I said there seems to be one thing ...
you've done fine in terms of where it's going to go to,
but once an instance of X becomes instantiated
then any other occurrence of X is going to be instantiated
to the same thing

Oh I see

E: In fact you wouldn't be able to hold here X being 1
and e being say 2 or ...

Right

E: Once that one gets an instantiation, which it did do -
I think that's the first one that gets an instantiation
isn't it? Now can you pick up from there now?

How you might ...

E: Its going to see if it can satisfy c(X) where c is 1

Yes. uhm.

Well it can't do that because c is 3.

Right ok. So where does it go now? what does it do?

Ehm ... well it would then go
it would then go back
and do b(X) again on this one
because that one failed
and it would try b(X) again,
its already tried that
one and that failed
so it would do that one
and it would um
it would go back to [mumble]
it would be disinstantiated if you like
so that X would come back again
and it would look for b of X
and again we have the [mumble] of f(X)
and f would be 3
so that will go to 3,
b would go to 3
and that will go to 3
and therefore all those will go to 3
and therefore we will be looking for c(3)
which it would find here
and therefore it would succeed ...
Ok when you call ...
2 \(a(X)\) with any bindings for \(X\)
3 which will be passed through into these two goals.
4 Um then you'll start off by substituting this sub-goal
5 as the major goal to be satisfied
6 passing the bindings through there,
7 so you're going to get carry down.
8 Then you're gonna pass the bindings across
9 so \(d\) is the first subgoal to be tried
10 so that's gonna become the new goal.
11 You then search down here for the first value of \(d\).
12 \(D\) is ...
13 the first value of \(d\) is 1.
14 \(1\) will be passed up through there.
15 If a wasn't instantiated to anything
16 or \(a(X)\) wasn't instantiated to anything
17 that 1 would be reflected back up through the chain.
18 You will have satisfied this goal,
19 so it will go on to the conjunct,
20 with \(X\) unified to 1.
21 It finds \(d(1)\),
22 that goal satisfies,
23 so we've satisfied
24 this sub-goal
25 and we come on to satisfy this one now
26 with \(X\) still unified to 1.
27 There's no match in the database for \(X\) being 1,
28 or \(c(X)\) being 1,
29 so this goal fails.
30 We then have to backtrack
31 and try and find an alternate solution for \(b(X)\),
32 which means throwing away what we did before
33 apart from the placemarkers ...
34 re-evaluating \(b(X)\)
35 and finding an alternate substitution for \(X\).
36 We tried it here,
37 so we try this one now
38 so \(X\) becomes unified to 2
39 so there's \(e(X)\),
40 try and find an \(e\) of 2,
41 there is one,
42 so it succeeds,
43 \(2\) goes across.
44 \(C\) of 2 fails again
45 so we backtrack into this thing.
46 Try and find an alternate solution for \(d(X)\),
47 there isn't one.
48 That goal will fail,
49 that goal will fail
50 because there is only one clause for 'a' ... No!
Clause 1 - a(X) if b(X) and c(X),

uhm I think I'd probably start

and look down at the facts

to see what might be instantiated.

Well c is going to be instantiated by that fact c(3).

Clause 2 b(X) if d(X) and e(X)

uhm and are both d and e are going to be instantiated

and d will be instantiated by this one first.

Uhm and 3 b(X) if f(X) and f(X) is instantiated.

Uhm I'm getting a feeling that if it's asked to backtrack

it will get the second ones for b for d for e

but if not it will just give one result.

Now what's the result going to be?

Well it's going to set up sub-

... suppose we give it the goal ?-a(X)

it'll set up the subgoal b(X)

and in trying to satisfy that it will take clause 2 here

and then try and satisfy another subgoal d(X)

and it will take the fact there

to satisfy it

and succeed on that part of the subgoal

and it'll try and satisfy e(X)

and again it will take the first clause for e(X),

satisfy that and succeed.

So uhm - ah it will only satisfy the whole

um goal b(X)

if the X's are the same.

I didn't notice that before.

In that case they are.

This is a 1 and a 1.

Uhm so it's satisfied the b(X)

if X is instantiated to 1

and now it tries to satisfy c(X)

and it looks down looking for clauses for c

and we get c(3)

so it's going to fail

because X had to be instantiated to 1 there.

So we have to backtrack

and try and satisfy b(X) again.

So we've failed on that clause,

we'll have to try on this clause -

hang on do we backtrack there

or can we backtrack to ...

uhm I'm not sure whether we backtrack

to the last choice it took on one of the sub-goals

um ... Now I think,

no because it's gone all the way through b(X)

I think we have to backtrack to

the last choice it took with b(X).

So um if we do that

we take the second clause

then we've got b(X) if f(X)

and trying f(X) as the subgoal now,

we get f(3).

So that is satisfied
58 with X instantiated to 3.
59 Now we try the subgoal \( c(X) \)
60 and find the fact down at the bottom \( c(3) \)
61 so that is satisfied
62 with \( X \) instantiated to 3.
63 So we have \( X \) the same for both.
64 So \( X \) has been instantiated to 3
65 and satisfied the goal \( b(X) \).
MSC. COOMBS AND STELL BACKTRACKING EXERCISE: SUBJECT 5

1 It'll take a(X) ... here uhm it will try to
2 find conditions in ... I suppose the body of the
3 clause here that will satisfy a(X).
4 So it will look first at b(X) and see if it
5 can instantiate or find anything such that
6 or ... b(X) is satisfied and then the same with c(X).
7 Now you've got a variable there X
8 and the same variable here ... uhm ...
9 I don't know I would have to try it out
10 to see if it worked.
11 I assume it would, I don't know.

E: Ok can you try to mentally work it out
then. You've come in with a(X) so the first
subgoal is b(X). Where does it go now? What
happens?

12 Well it will search down the database
13 to see if it can find any instance of X.
14 I don't know if this X is meant to be uhm
15 a variable or a constant ...

E: it's a variable

16 a variable uhm such that b(X) occurs.
17 Now this is our database here.
18 I think it's gonna probably going to fail
19 so that won't be satisfied
20 so I don't think it will ever bother
21 with the second one
22 as that one hasn't been satisfied
23 so therefore it will return 'no'.

E: Ok right, fine. So that was assuming
that there were no actual instantiations

23 Right. Are these rules in as well?
24 [indic. Clause 2 and Clause 3]

E: Yes.

24 Yeah well obviously then we've got b(X)
25 and it will find there's a rule for
26 b(X) if d(X) and e(X). Now it can ...
27 I'm not sure how Prolog works,
28 but it can presumably, possibly
29 it will instantiate d(X) as d(1)
30 [indic. Clause 4].
31 I don't know uhm if it's a variable.
32 I presume it could pick up that value
33 so it'll take d(1) in there and
34 go on to the second part here.
35 Now X is going to have to be the same
36 value in both cases - it's instantiated as 1.
So 1 and 1 and that succeeds so it will come back to here
   [indic. Clause 1]
and X is gonna have to be 1.
Now it's gonna have to find c(1) because X here is instantiated the same,
so we look down the database - no c rule - which I missed out the rules last time -
and lands up at 3. No.
So it'll fail on that. So it'll then backtrack and see if it can find anything else for b(X).
So back at the rule here we have to get rid of the - wherever we were -
I don't know - I was on d, yeah, d -
I'm getting lost now ...
Yeah d(1) so we try 2 now - d(2)
and e is going to instantiate the same and it can, it succeeds, so, fine,
so b(X) comes b(2),
b(2) here, so now it looks at the c -
C is 3 - there are no rules for c so that fails -
right, so it'll backtrack through discounting all of those,
coming back to some pointer -
I'm lost as to which pointer I was probably at -
but presumably, it must be b(X) - uhm d(X) ...
Right, now I think we truly are stuck because I can't find any other values for d(X) which would satisfy b(X)
which would satisfy that one
   [indic. b(X) in Clause 1].
Uhm ... b(X)
   [in Clause 3]
well I haven't, yeah, right, uhm so you could try this rule now b(X)
if f(X), f(3) ... Huh, right, b(X) ...
so if that gets instantiated as 3,
b(3), and then c(3) ...
I can remember that one cos we've failed on that before,
c(3) right, so those have both been put at 3,
b(3) and c(3)
uhm, so er a(X) can take on the value of 3,
so it'll say 3 question.
And if you press -
looking at it I don't think you'll get any other value,
so if you press return you're gonna get 'yes'
if you press return, you're gonna get 'no'.

MSG: COOMBS AND STELL BACKTRACKING EXERCISE (CUT): SUBJECT 5
1 Right, it would take a(X) and try and see
2 if it could satisfy this first rule here, a(X),
3 and it would go to the first rule b(X) ...
4 um ... and try and satisfy that,
it would see \(d(X)\),
X could then be instantiated as 1,

so it would then stop with that value 1

I think that's right,
Um ... I'm not sure presumably that initiates backtracking

it then has to go on and satisfy this clause here.

\[\text{indic. } e(X) \text{ in Clause 2}\]

It'll stay with 1, yeah, so it'll take \(e(1)\) which it can do,
so that's now satisfied, \(d(1), e(1)\)

so come back to ...

\(b(X)\) is \(b(1)\) ... and er

for consistency's sake this \(c\) is going to have to be 1
so it's going to see if it can satisfy \(c\) anyway.

Um ... looks like we've hit a problem

anyway it's got \(c(3)\) and \(c(4)\)

[i.e. in the database]

can't do that, so that's going to fail.

right, so it'll backtrack to \(b(X)\)

and see if there's any other ways that it can resatisfy \(b(X)\)

which it can do, matching against this rule here

[i.e. Clause 3]

\(f(X)\) and \(f\) is 3.

\(a(3), b(3), \text{ and } c(3)\)

and finds \(c\) as the first one here,

so we're all right,

so we've now got um

\(X\) instantiated as 3 throughout

it'll return \(X\) equals ...

\(a\) equals 3 ... or \(X\) equals 3 ...

my mind's spinning ... Yeah \(X\) equals 3.

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

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

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

E: Right, ok. So a(X) to set you off ...

12 Right, so now I've got a(X) I'm a(3) if b(3) and c(3). Ahm ... Ok now what I'm hoping

[indic. Clause 2]

14 is that - this is getting a bit confused now ...

15 Now I'm not Prolog any more, but me being a little confused because I'm hoping that I can also instantiate b to the same thing ... what I just instantiated c to, so that ... so that I'm justified — I think I'm confused now, 'cos I'm doing it half in Horn clauses and half just in my head, saying what needs to be the case for a to be the case.

23 And I kinda set about doing it as if I were resolving it and only doing half 'cos I'm not adding the negation of what I'm trying to prove.

E: Right. What I'm interested in is why you have picked up an instantiation — or at least how you've picked up the instantiation of 3 for X.

26 Being myself?

E: Well that's how I think you did it!

27 It is isn't it! What I did was in just sort of English, not really in logic — I was trying to find the same value ahm of which it's true that both b of that value and c of that value, so since my first rule

[indic. Clause 1]

32 mentions b and c there was no b immediately available to me

[indic. facts]

34 so I thought oh, ok I'll go and investigate that later — maybe I'll have to

[indic. Clause 2]
36 use this rule, or this one
     [indic. Clause 2 and Clause 3]
37 but there is a c, I can get my c to be
38 true immediately and I'll be able to get a
39 - I'll be able to use a(3) if I've got c(3)
40 and then kind of on my agenda of things to be done
41 I can also establish b(3).

E: Ok, now that's a very sensible way for
you to proceed, everything that you've
said is true - i.e. that the only possible
instantiation is c(3). But can you take me
through how Prolog would set about doing it -
the mechanism - nothing to do with logic.
What's the mechanism that would allow Prolog
to pick up any instantiation at all for X,
and what Prolog wouldn't be able to do is
jump to the end of there and scan the
database. It wouldn't do it that way. Now can
you tell me how it would do it?

--- pause ---

42 Uhm ... I think how it would do it is ...
43 well I suppose how it would do it ...
44 All it's got to play with are the facts
45 and a new fact which it's trying to say not(a(X)).
     [i.e. the query]
46 And it's going to try and ... no.
47 Now it's apparent to me that
48 I don't really understand how it's going to do it.
49 Cos what I want to say is it's going to try
50 and resolve the new fact with what it's got
51 but it's also obvious that the new fact doesn't,
52 first off, resolve with anything.

E: I would like - it would be easy for you
I think to make distinctions between the
truth of the logic of the description of
Prolog, and the mechanism that actually
executes the logic. Do you see what I mean?

53 Yeah ... 

E: So, for example, what that
would involve is saying ok, we come in with
a(X) and we can prove that a(X) is true
whatever a(X) gets instantiated to, if we can
prove b(X) and c(X). Now this is your goal
stack, right?

54 I see that it goes here to No.2
55 because it's the first clause that mentions of b(X)
56 on this side,
57 or just as a fact on its own.
58 So then I would say that it looks at this rule
59 and says b(X) is true if these two things are true
so ... it adds those 2 things
and the first of them to be addressed is \( d(X) \).
First \( d \) it comes to is this one

\[ \text{[indic. Clause 4]} \]
so it tries ... it tries instantiating \( X \) to 1,
puts 1 in for \( X \) here
then I would say backs up the goal queue
so the next thing it has to worry about
is whether that's a valid instantiation,
also here

\[ \text{[indic. } e(X) \text{ in Clause 2]} \]
since it had taken off \( d \)
and the next thing under was \( e \).
Looks down for some \( e \)'s.
Here's an \( e \) - it's 1, so far so good.

E: Why is that good?

Because in order for \( b(X) \) to be true
both \( d \) and \( e \) have to be true of the same \( X \)
And so my strategy right now
is to be trying \( X \) as 1.

So \( X \) is 1. Ok.
So having facts for \( d \) and \( e \), then,
we're safe to say \( b(X) \) is true when \( X \) is 1,
so now we've backed up to here

\[ \text{[indic. } b(X) \text{ in Clause 1]} \]
We've taken care of this bit.
Now er have to see if \( c(X) \) is also ok when \( X \) is 1
in which case, together, we can say yes,
there is an \( X \) for that that'll make that one true.
But there isn't anything really justifying to say \( c(1) \).
So at that point, being Prolog, not myself,
I would say well, that didn't work for \( X \) is 1,
but I'm not going to give up still on this ...
Sorry I'm not ... where have I backed up to?
Sorry I'm not gonna give up on this rule yet
there might be another \( b \) that would still
satisfy this condition for \( a \).
Ahm ... ok.
Look for another \( d \).
I'm remembering that last time I tried 1,
so now I'm ...
I just did something in my head there which I didn't say
but Prolog would do it ... uhm ... no
and I think it would work ... scrap that.
What I would do being Prolog -
the last thing I tried was a value for \( e \)
which I had been directed to by this value for \( d \)
at the time I was trying out \( d \) being 1
and that caused me to try out \( e \) being 1.
Uhm ... that didn't work so what I hesitating with now
is where I back up to ...
I'm not sure ... I guess I'm sort of tempted to say
well Prolog just makes a stab at saying \( e \) is 2
but it's not the same \( X \)
110 so it just doesn't work.
111 So back up to d, so ...

E: that's exactly right ...

112 Why I stopped myself is 'cos I just
113 did that in my head as being kind of obvious,
114 but I suppose it's not obvious
115 it is actually a step.

E: It is a step.

117 Right. Why I hesitated is
118 'cos I realised I had done it in my head
119 for a slightly wrong reason
120 which was sort of what was in my mind was
121 Oh, X is 1 now so I was scanning through
122 looking for another e expression involving X is 1
123 and that is in fact the criterion
124 by which to work out in the end
125 for my purposes but I think
126 it's the wrong kind of orientation.
127 I think being Prolog
128 first he'd look for an e
129 and then work out if he can get an X is 1
130 for an e
131 whereas I just said instantly
132 oh there's an e but it's 2
133 so I won't even mention it to you.

E: Ok. But the crucial point to remember is
that the backtracking starts from here, as it
fails to redo e(1) ...

134 Then I would backtrack to d -
135 I've already tried this one
    [indic. Clause 4]
136 so I try instantiating X to 2,
137 now I'm looking for an e with X being 2.
138 I look at this one again
    [indic. Clause 6]
139 even though I've looked at it once before
140 because now I'm looking at it for a different d,
141 but it's no good.
142 I look at this one -
143 yes, it is good,
144 so I've satisfied both that and that
145 so I'm safe to say b(X) is true if X is 2
146 so I've got that bit.
147 If it's true I have to prove a c when X is 2.
148 Here's my only c,
149 but it won't do, so that's that.
150 Do I have any other c to try,
151 no I don't.
152 So I have ...
153 so that strategy wasn't any good either,
154 but I'm still at this rule
still trying this one -
this rule
[indic. Clause 2].
I look if there's any other way
I can try out this rule
which would have to start with
any other instantiation of X for d.
I've already done that one.
I look down the list -
there's nothing else.
So that's the point at which I can say
well, I'll dispense with this rule,
for the purposes of solving b in here.
Still I haven't given up on Clause 1 -
there might be another way of solving b.
So I look down the list and yes there is.
There's this one here. Uhm ...
So at this point I've taken off all the things
that were generated from 2
out of my list of things to do,
but I am still interested in this possibility.
[indic. Clause 1]
So I still have b and c on my list of things
but I've taken - I've exhausted all the d's and e's.
So now I try out this way of making this true.
I have to find an X of which f is true.
I go down my list looking for f's -
I find one here.
F is true when X has value 3,
that's all that has to be true.
There's only one condition, so fine.
There's a possible b when X is 3,
go back up here - Ok got that bit when X is 3.
[indic. Clause 1]
Try out c when X is 3 -
I go down the list.
Here's a, c - it's a good one - it works.
So now I've satisfied both those bits -
they both have to be true at the same time
and they can be.
Same X in both cases is 3.
So I know that when X is 3, a is true,
so at that point I would return 3.

MSC. COOMBS AND STELL BACKTRACKING EXERCISE (CUT): SUBJECT 6

Ok, uhm, first thing is er
look for a rule mentioning a(X)
and find one here
[indic. Clause 1]
and the first thing I have to prove is b(X)
I look for a rule telling me how to prove b(X).
I find this one,
first thing it tells me to do is look for d(X)
so I look for a rule mentioning d(X)
This is the first one I find
[indic. Clause 4]
10 and instantiate d to 1 in here
   [indic. Clause 2]
11 go past the cut ... uhm
12 e(X) - try and prove that.
13 X is now 1, come down to here,
14 fine - that holds
   [indic. Clause 5]
15 so the instantiation of X as 1
16 I've satisfied both of these
   [indic. Clause 2]
17 therefore I've satisfied b(X),
18 go back to here,
19 I've satisfied this part
   [i.e. b(X) in Clause 1]
20 again, with X as 1 ...
21 look for a rule mentioning c,
22 find my first on here
   [indic. Clause 9]
23 doesn't do any good.
24 Second one there again,
   [indic. Clause 10]
25 doesn't do any good,
26 look for some more,
27 there aren't any more.
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
   [indic. Clause 2]
35 to prove b,
36 I have to prove this, do d(X) first thing ...
37 uhm this was my first one
   [indic. Clause 4]
38 this one I've already used ... uhm ...
39 so - cos that was the failure here
   [indic. d(X) in Clause 2]
40 so I'm allowed to skip past that one,
41 instantiate d to 2.
42 Try - d is 2 here
   [indic. Clause 2]
43 go past, e,
44 look for something e
45 I want e to be 2.
46 I find it here.
47 Find I've satisfied b,
48 now I have to satisfy c with X as 2.
49 This doesn't do it
   [indic. Clause 9]
50 this doesn't do it
   [indic. Clause 10]
51 Nothing else for c.
52 So I failed again here.
53 Backtrack again ... uhm ...
54 backtrack back to b,
55 look for another way to do b,
56 come to this rule again
   [indic. Clause 2]
57 look for something for d -
58 tried that
   [d(1)]
59 tried that
   [d(2)]
60 nothing else for d,
61 so I have to go right out.
62 There's no other way of trying this rule.
63 There's another rule I can try, uhm this one ...
   [indic. Clause 3]
64 Uhm, told me to go and find an f(X),
65 look for one,
66 find one,
67 Now X is 3, uhm ...
68 go back to here
   [indic. Clause 1]
69 X is 3, ...
70 look for c(X) with X as 3.
71 So down my list looking for c,
72 find it here
73 c(3), both of these are satisfied,
74 so I've satisfied a(X) when X is 3.

E: right ...

75 But I think I did something wrong
76 because I didn't do ...
   [indic. !]
77 there's something I didn't do!
78 With the cut there!
MSc. COOMBS AND STELL BACKTRACKING EXERCISE: SUBJECT 7

1 To satisfy a then there's only one clause that matches a
2 in the list of clauses we have 1 to 9
3 so it will match with this clause
   [indic. Clause 1]
4 which will convert the goal of trying
5 to satisfy the goal of a into the goal
6 of trying to satisfy these 2.
7 The first of these goals
8 which is the one Prolog will pick to try and satisfy
9 putting this out to satisfy later
   [indic. c(X) in Clause 1]
10 is b, and b can be satisfied twice in these 2 clauses.
11 So it will pick the first one.
12 So it will now try and satisfy b
13 which will create these two subgoals
   [d(X) and e(X)]
14 still keeping in mind that c is to be satisfied later on.
15 The first of these is d -
16 so there's now a stack of goals which is d then e then c.
17 So it'll try and satisfy d and there's two possibilities
18 two ways that that can be satisfied,
19 and it'll take the first of those
20 and X will be instantiated to 1.
21 So it will have satisfied d and will come onto
22 the next goal in the stack which will be e.
23 This time it will be e(1)
24 because the X was instantiated to 1
25 when it satisfied d
26 so now it will try to satisfy e(1)
27 which in this case there's only one case that matches
28 and it does - d(1) and e(1) match.
29 So the whole of b will succeed,
30 X will have been instantiated to 1,
31 which means this goal here which was the original quest
32 to try and satisfy this, will have been satisfied
33 and X will now be equal to 1 in here,
34 so we move on to the final goal to be satisfied
35 which is c,
36 and it'll be trying to satisfy c(1)
37 since these two X's are the same.
38 So it'll try and satisfy c(1)
39 and unfortunately there are no clauses
40 in here which satisfy that,
41 so that will fail.
42 So it knows that X instantiated
43 in this goal here being 1 doesn't work,
44 so it will backtrack and try to satisfy b a second time,
45 b(X), having remembered that the first time
46 it matched on this clause
47 it will now try and resatisfy b(X)
48 using the second clause.
49 There's only 2 of them,
50 so it will use this one
   [indic. Clause 3]
and it will try and satisfy \( f(X) \).
There's only one of those that works, so it will satisfy - succeed, sorry - and \( X \) will be instantiated to 3. So that goal will succeed and it will try and satisfy \( e(3) \)

\[ \text{[NB: } e = 3?] \]
since \( X \) was instantiated to 3.
There is no \( e(3) \), no clause that matches that, so this will fail. Ok so where have we got to now? Still trying to satisfy \( b(X) \).
Ok. It knows this one didn't work

\[ \text{[indic. Clause 2]} \]
it nows this one

\[ \text{[indic. Clause 3]} \]
didn't work the last time and there's - oh sorry where am I?
I've got lost ... it tried to satisfy \( d \) ...
did it try ... oh ...

E: ok, it tried these with 1 and didn't work, it failed and we're setting off to try and satisfy \( b \) some other way.

Oh right, ok, thanks.
So it matched on this one

\[ \text{[indic. Clause 3]} \]
which created the sub-goal \( f(3) \)
\( X \) was instantiated to 3 and \( e(3) \) failed.
So it can go back now, back here aren't we?

\[ \text{[indic. } f(X)\text{]} \]
Oh dear, I'm completely lost now.
It matched on that one

\[ \text{[indic. Clause 3]} \]
and \( f \) was instantiated to 3
so, sorry, yes, right, right.
It matched the \( b(X) \),
the goal \( b(X) \) that it's trying to satisfy,
created the subgoal \( f(X) \), succeeds,
\( X \) is equal to 3,
\( b(X) \) succeeds,
so it then tries to satisfy \( c \) with
\( X \) instantiated to 3,
tries to satisfy \( c(3) \)
and it succeeds,
and it should print out \( X = 3 \).

MSc. COOMBS AND STELL BACKTRACKING EXERCISE (OUT): SUBJECT 7

1 The difference is there's a cut here.
2 Let's see if I can get the hang of it this time!
3 So presuming a query \( a(X) \), ok
4 so it calls this

\[ \text{[indic. Clause 1]} \]
5 as a goal
6 which will create 2 subgoals \( b \) and \( c \).
7 It tries b which means that it must satisfy d,
8 so it will go through and produce,
9 it will match with this clause
    [indic. Clause 4]
10 and X will be instantiated to 1.
11 So this will succeed
    [i.e. d(X) in Clause 2]
12 go over the cut and then it will try for e
13 and this time it will be e(1)
14 because the two X's share,
15 that will be satisfied
16 and it will print out X equals ...
17 Oh! hang on - that will finish the b goal
18 then it will go for the c goal, c(1)
19 which doesn't exist here
20 so that fails.
21 So it will backtrack.
22 Now let's see if I can get the backtracking right.
23 First of all it will try the c
24 but it must try c(1)
25 because that was instantiated up here
26 and there's no way that can be satisfied,
27 so that's no good,
28 so it will then try b again.
29 This time it's coming in here
    [indic. e(1) in Clause 2]
30 And it's now looking for an e(1)
31 because the l's set up at this stage
    [indic. d(1)]
32 so it'll try for the e(1) - no sorry
33 it'll try and re-satisfy e(1) which it can't do
34 there's only one e(1) clause -
35 the 1 was instantiated here
    [indic. d(1)]
36 so that will fail.
37 But now because it comes to the cut,
38 it won't traverse any further back in these goals
39 but the whole of the b clause will fail
40 but not only that,
41 but this clause
    [indic. Clause 3]
42 will fail as well.
43 So the b hasn't succeeded
44 and there's no way the b can succeed
45 therefore a can't succeed,
46 so it will just print out 'no'.
Right well, the first ...
I would look at the first rule
and er say that uhm it [mumble]
so the first rule says \( a(X) \) if \( b(X) \) and \( c(X) \)
c of \( X \), so -
[pause]
that doesn't seem to ...
well nothing in the database will match that ...
nothing in the rules ... er facts list
seems to match with that,
so it goes on to the rules.
\( b(X) \) is \( d(X) \) and \( e(X) \) -
nothing matches there either.
Oh! In fact that doesn't work. Uhm ...
In fact only rule 1 applies really ...
only rule 1 has \( a(X) \) on the left hand side.

E: Right. So it comes in here
[indic. Clause 1]
and as you say it sets up the sub-goals
right. How does it go about proving \( b(X) \)?
Which is it's ... the next thing it has to do?

17 Uh huh. Yes, right ... ok ...ahhh ...
18 Yes well, uhm ... so it goes to rule number 2
19 and er and it finds \( b(X) \) if \( d(X) \) and \( e(X) \) ...
20 uhm ... - pause - and uhm [cough] and uhm
21 I don't think it'll get very far with that.
22 So then it goes on and it takes the second rule,
23 \( b(X) \) if \( f(X) \).
24 So another subgoal is set up, \( f(X) \).
25 That's right, so uhm,
26 then it scans this part of the program
[indic. facts]
and ends up with a value of 3,
3 instantitated to \( X \),
so \( f(X) \) succeeds and so rule 3 succeeds,
so er \( X \) is instantiated to 3
and so the subgoal succeeds up here
[indic. Clause 2]
32 sorry ... er ...

E: there ...
[indic. \( b(X) \) in Clause 1]
33 Yes er ... well although it's gotta have ...
34 er ... yes, so that this subgoal succeeds
[indic. \( b(X) \) in C1]
35 but um ... 
36 Yes I suppose that \( c(X) \) would succeed too
37 because um the same value here 3
38 HUM!!!!!
39 We'll go round in circles!
MSc. COOMBS AND STELL BACKTRACKING EXERCISE (CUT): SUBJECT 8

1 Presumably you're asking a(X)?

E: Yes.

2 Ok. Uhm so it goes to the first rule
3 which matches the head of the first rule
4 a(X) with the problem a(X),
5 so it sets up hum the rule says a(X) if b(X) and c(X).
6 So it sets up b(X) as a subgoal and uhm
7 b(X) matches to the second rule,
8 so it then says that d(X) is a subgoal
9 and attempts to satisfy d(X) uhm
10 so again it looks along the list of rules and facts
11 and finds er that well
12 the first d it comes across,
13 it matches first on d(1)
14 so X is instantiated to 1
15 and then er, so yes, so it's succeeded in ...
16 no that's right ... uhm, so ... um

[pause]

17 So uhm it then attempts to satisfy
18 the rest of the rule b(X) - is that right?
19 so it comes across the cut sign, and er ...  
20 hopes I think it hops over that because the cut is only
21 ... applies when you're backtracking through it,
22 so it hops over that, and then tries to satisfy e(X)
23 so then it looks down the rule list
24 and finds again e ... that matches with e(1)
25 so X is instantiated to 1. Yeah.
26 So then the whole of the right hand side of rule 2
27 uhm actually succeeds,
28 so b(X) succeeds with e,
29 with X instantiated to 1.
30 So first subgoal of first rule succeeds as well
31 so then it starts looking for c(X) -
32 is that right, am I doing ok?

E: Yes.

33 And then ... so it scans down again
34 and it finds,
35 is unable to satisfy the second subgoal
36 with X instantiated to 1,
37 so the whole rule fails.
38 I'm doing better than I did last time!
39 So the whole rule fails
40 and so it undoes the instantiation of X to 1
41 and er ... and X becomes uninstantiated
42 so then it um it tries ... it then attempts
43 it sets up b(X) again as a sub-goal.

E: It doesn't ... the whole rule hasn't failed yet ...

44 Ah!
E: That failed
[indic. c(X) in Clause 1]
and that initiates backtracking right. So this now
[indic. b(X)]
sets off again, you know ... We pick up from here,
so we go back to the last decision point to get
a satisfaction for b(X).

45 Uh huh.

E: So can you pick up from there? Go back
to the last decision point.

46 Where did I get to? Yeah, so e(X) has failed,
47 right, that's what I said didn't I?

E: No, c(X) has failed which means you're
backtracking to b(X) again.

48 Ah, I see.

E: So where do you go?

49 Yes, well ... e(X).

E: That's right!

50 It goes back from the right ...

E: But what happens to e(X) now?

51 Yeah ... right ...

E: Given that X is instantiated to 1 at
the moment ... it's been instantiated.

52 Well it goes back through the fail port
53 doesn't it ... the box ...
54 back through the fail port ...

E: It will do in a moment. There's one other
thing it does first. Its come back to here
[indic. e(X) in Clause 2]
and it's got e(1).

55 Yeah

E: It'll check the database to see if
there's any other way of satisfying e(1)
first, and because there isn't, then it
fails, ok, and as you say it comes in
backwards through the fail port.

56 How could it have another way of satisfying ...

E: Well it hasn't got another way of satisfying e(1) ...
57 [interrupt] Well how could ...

E: In this program, there might lower
down be a rule that says another way of
satisfying e(1) is such and such and it will
check that possibility first. Or if you just
had another occurrence, if you had another
e(1) down here, then it would succeed and
carry and do the whole thing again.

58 Oh I see, of course,
59 because it can't remember where it's been.

E: Well it can't distinguish between one
example of one thing and another example of
the same thing.

60 Hmmm ...

E: It would treat them as two separate
objects if they occur at different times in
the database. But sensibly speaking there may
be programs where it is possible to resatisfy
the same thing via a different method and it
would have to check all possibilities of
resatisfying that goal before it carries on
to do anything else.

61 Hmm mmm ...

E: So having done what happens to be a
redundant exercise in this program, it now
backtracks through the cut. So what happens
now?

62 Right. So, yeah, so
63 it hasn't found any other way of satisfying 1
64 so ah, so when it comes to the cut
65 it doesn't go through d(X),
66 it bypasses it and goes to the head of the rule.
67 Is that right?
68 And ... er ... so hang on,
69 so it ... and that makes it fail, I think -
70 does it? - it makes b(X) fail because of the cut.
71 Uhm ... so that means automatically b(X) fails.
72 Its unable to get a value for X,
73 for b, whatever,
74 so it doesn't bother to look anymore
75 for another way of satisfying b(X).
76 So ... uhm.

E: So the fact that this clause is there ...
   [indic. Clause 3]

77 So it doesn't match here
   [indic. Clause 3]
78 because it doesn't look I suppose.
79 Just fail.
80 If it hadn't been for the cut,
81 it would have gone along
82 and tried to resatisfy,
83 satisfy b(X) by putting f(X) as a subgoal.
MSc. COOMBS AND STELL BACKTRACKING EXERCISE: SUBJECT 9

1 You've given X as a goal
2 so it'll start at the top of the database
3 and it would say X if b(X) and c(X),
4 a(X) if b(X) and c(X)
5 and so it would look at the subgoal
6 and it would see b(X) if d(X) and e(X) ...
7 and so it would look at the subgoal and
8 go away and look for d(X)
9 and it would find d(1)
10 so it would bind X to 1
11 and - does that bind X to 1 for the whole clause?

EXP: Yes.

12 It does - so you get d(1), e(1)
13 and it goes through to see if it can find an e(1)
14 and it can,
15 so it succeeds on Clause 2 -
16 it's found a b(X) - does that feel right?
17 b(1) if d(1) ... yeah yeah ok
18 so it goes away and pops back up here
   [indic. Clause 1]
19 and looks at the next subgoal
20 and it goes blot, blot
   [indic. past Clause 2 and Clause 3]
21 Are these all currently instantiated to 1?
   [indic. variables in Clause 1]
22 They are, aren't they,
23 so they're going forward to look for a c(1)
24 and, surprise surprise, it doesn't find one!
25 So it's got to backtrack
26 and look for further instantiations of b
27 and it finds that b(X) if f(X)
28 so it goes through and looks for an f
29 and it finds an f as 3,
30 so it succeeds with b f as 3 ...
31 and so that's that instantiated to 3,
32 a(3) if b(3) and c(3).
MSC. COOMBS AND STELL BACKTRACKING EXERCISE: SUBJECT 10

1 Look for a(X),
2 so it has to satisfy these goals
   [indic. Clause 1]
3 taking b(X) ...
4 uhm so it must go to here
   [indic. Clause 2]
5 and this depends on ... if d(X),
6 so it looks for an instance of d(X)
7 so it instantiates it to 1
   [indic. Clause 4]
8 so that's instantiated to 1
   [indic. b(X) in Clause 2]
9 that's instantiated to 1
   [indic. b(X) in Clause 1]
10 and that's instantiated to 1
   [indic. a(X) in Clause 1 and the query]
11 So X is 1.
12 Then it waits for you to ask again.

E: Right, ok. In fact that's only half the solution. What you've done is you've got ...

[Interrupts]

13 Oh yeah, sorry I haven't finished,
14 and it goes on ...
   [indic. c(X) in Clause 1]
15 I'm not sure whether it then
16 presumes that c is 1 as well
17 if you're, you know ... to finish it.
18 I would have presumed it would
19 once it had found that
   [i.e. that X is 1]
20 then it would ...
   [pause]

E: And then it would finish?

21 Mmmm.

E: Right, ok. You're right to say this matches that ... [explanation of calling d and e in as subgoals above c ... looking for instantiations for d] ... now X is 1 as you quite rightly said ...

22 Sorry I completely left out e didn't I!

E: Ok, all these become 1 now. Can you pick up from there now you realise you left out e?

23 Well ... I would have just gone on
24 and said that goal is satisfied
   [d(1) in Clause 2]
25 and e ... it picks up e,
26 the first case of e(1)
27 and then goes back
28 satisfying all its goals up there.
29 Uhm ... then it would stop
30 and wait for you to ask again.

E: Ok, so you got d(1) and then you checked
that you'd got e(1) to come out here
[i.e. out of Clause 2]
so you've proved b

31 Oh yeah ... 

E: ... with X instantiated to 1.
So what happens next?

32 Oh that's what I wasn't sure
33 whether it would actually just accept
34 that b(X) is ... was X
35 and be happy
36 or whether it would actually go through again
37 looking for ...

E: Right. What's the difference between
this clause
[indic. Clause 2]
and that clause?
[indic. Clause 1]

38 Uhhmm. In what way?

E: In their structure ...

39 Uuhmmm ...

E: Because when we did this one
[indic. Clause 2]
we found an instantiation for d, and then
checked there was one for e. Why wouldn't we
do that here?
[indic. Clause 1]

40 Well, I wasn't even sure there
[indic. Clause 2]
41 whether in fact it would actually do that or not.

E: Right, well let's go through and check
it out. We agree on d ... [so Clause 2 succeeds] and
we come out with X instantiated to 1 ...

42 So in fact it does actually, it doesn't just
43 take it for granted
[indic. e(X) in Clause 2]

E: That's right, it does have to check.

44 Oh well in that case ...
E: So can you pick up?

45 So then it has to try that
46 and in fact when it comes down to c
47 it has to backtrack
48 because it can't instantiate c to 3,
49 so if it has to backtrack that fails
   [indic. c(X) in Clause 1]
50 so it goes back to try
51 and see if it's got another choice point at uhm b
52 and b ... it had gone to there
   [indic. Clause 2]
53 well it's ... I'm not sure if it would go back
54 and try that again to get
   [indic. Clause 2]
55 or whether it would go straight on to there
   [indic. Clause 3]
56 Uhm ...
57 I'm trying to think back what I was doing before.
58 I'm trying to find the last choice point ...
59 from here ...
60 uhm in fact the last -
61 presumably - would it go back
62 to the very last choice point there
63 and look for another value for e,
64 or it finds 2
65 and then it would try to resatisfy d(2), ...
66 and it tries
   [indic. c(X) in Clause 1]
67 and then it would fail again
68 because it can't find a c(2).
69 So it would go back again
70 and it can't find another e,
71 so that failed,
72 so then it would go on to the next b(X)
73 and get f
74 and it will find an f
75 so it will go back trying to find a c(3)
76 and then it can satisfy that,
77 so it'll come back with a as 3.

MSc. COOMBS AND STELL BACKTRACKING EXERCISE (CUT): SUBJECT 10

1 So it's trying to satisfy the goal a(X) that
2 will be satisfied - proved - if I can find
3 that b(X) - do you say 'b of X' - yeah - and
4 c(X). So first of all look for b(X)
5 and I find b(X)
   [indic. Clause 2]
6 and b(X) is true if d(X)
7 and e(X). So I look for d(X) - sorry can I do
8 that to remember where I am?
   [putting thumb as placemarker on b(X) in Clause 1]
9 and that's a terminal so that's 1,
10 so I hold that as 1 for a minute,
11 and then try to satisfy e(X) and
12 e(X) is a terminal so that is 1, so at the
moment um I've satisfied b(X) and I go back
and try to satisfy c(X) and c(X) is a
terminal, so it's 3 - pause - but that's not
equally, sorry ... a(X) if b(X) ... and at the
moment b(X) ... Oh I've lost the thread of
where I am, sorry, b(X) is true if d(X) ...
       [indic. Clause 2]
and so I ... that's not ... that's
instantiated to 1
       [indic. b(X) in Clause 2]
that's instantiated to 3
       [indic. c(X) in Clause 1]
So it's not true, so I should be backtracking.
So I see if there's any other
way of doing c to get 1,
but I can't so at this point I go back to resatisfy b,
but I can't resatisfy b because the cut's there,
so I have to fail.
Sorry if my terminology's all wrong!
We call ... it's calling a(X)
so that matches up with rule number 1,
and er which ...
so that mat- so it calls a(X) so that ...
so a(X) matches here
[indic. Clause 1]
and it attempts to satisfy this
[indic. b(X), c(X) in Clause 1]
by calling b(X) um
which ... so it comes back out
and finds b(X), right.
Uhm ... it then it finds b(X) in Number 2,
uhm but it finds um ...
to satisfy b(X) it has to satisfy d(X) and e(X)
so it calls d(X) and it goes through
and number 4 satisfies d(X),
so X gets instantiated to 1.

Right? Uhm, which then instantiates (b(X),
X becomes 1 there
[indic. head of Clause 2]
and 1 here
[indic. body of Clause 1]
and here
[indic. head of Clause 1]
so that's the instantiation that takes place.
So it's at this stage it moves on
to see if it can instantiate -
if it can satisfy e(X)
and it finds that it's number 6 - that e -
own do we do that first
and come back to e(X) ...
right, it would, no it would have to do this first,
do e(X) first
so it would get e(X) instantiated to 1 ...
uhm but ... with X being a variable there ...
I don't think that matters.
I think the 1's ok there as
[indic. e(X) in Clause 2]
well as 1 being ok on previous ones.
[indic. higher up the program]
So er ... it ... then would satisfy that rule
[indic. Clause 2]
so it's got b(X),
it goes back and tries for c(X)
and it's X becomes instantiated to 3 ...
Is this ...
Am I getting confused with these X's?

E: I think so, a bit, yes.

Because, ah right,
because it's the same variable isn't it?
E: That's right, it's all the same variable.

OK, right, in that case it can't do that then.

E: Right

It, uhm ... if it's all the same variable right ... look here, it takes ...
Can I start again?
a(X), b(X), c(X), so it comes in a(X) and it goes down and it matches here
[indic. Clause 2]
which calls b(X)
which calls d(X) which takes 1,
uhm b(X) uhm which is 1 ...
so at this point it fails
when it finds that c(X) is 3

Uhm, ok.
Cos this, cos this was a 1
this was a 3.
So it would come back with 'no'.

MSc. COOMBS AND STELL BACKTRACKING EXERCISE (CUT): SUBJECT 11

a(X), right, well it would look for a rule which it would match up with the first rule, the first clause there, er, a(X) which would then attempt to satisfy the sub-goal b(X), it would come to the second clause here and it would attempt ...

it would find that it could satisfy that if it could satisfy d(X) um ... so it would find that with the first fact here,
d(1), so X would be instantiated to 1 throughout - it would then hit the cut. And it would pass through the cut and hit e(X) so you're looking in the database for e(X) and it would come down and satisfy it with e(1).
It would then, because the cut is there ...
hem ... I have to think about this for a minute ...
hem ... will I just talk through what I'm thinking about?

E: Yep.

Hem ... I know that cut immediately cuts the flow of satisfaction, so it would cut back to the parent goal. What I'm thinking about here is whether e(X) can be resatisfied with e(2) here before it gets to the point where it fails. I think e(X) has to fail first before it starts to go back as soon as it hits the cut, what that ...
I think it has to set a value e(2)
before it eventually ... before it fails
the flow of satisfaction,
but back to the parent goal - right?
Which then makes the system committed
to all previous choices it made between
the parent goal and the cut ... Um
so that's a problem.
I'm not sure if that would do e(2)
before it would fail,
but I would say if it failed it would cut back ...
hem ... that would mean that would go right back ...
it's saying well once you've come to this point,
that's that's it ...
you've ... that's the choice you've made,
and you're committed to that,
so you cut back to here
        [indic. b(X) in Clause 1]
and it would then do c(X),
but if it could satisfy it,
        [Ambiguous]
but if it ever failed, .
that c(X) would be cut right out.
Ok it's trying to prove a(X)
to do that it's got to prove the subgoals b(X) and c(X).
So it looks to the database -
rather, first of all it looks to the database to see
if a(X) is there and it's not.
There's no functor in the database.
So it tries to prove b(X),
it looks to the database,
there's no functor b in the database
so it looks to the rule b(X), rule number 2,
and finds that to prove b(X)
it has to prove d(X) and e(X).
It looks in the database for d(X) and finds d(1)
so X is instantiated to 1
in both the d subgoal and the e subgoal.
Then it looks to the database to see if e(1) is there
and finds fact 6, that it is,
and them uhm, this instantiates b(X)
and the clause to value 1 ... uhm
It then tries to prove, I think,
rather than backtrack
and trying to prove d and e again,
I think it then goes to try and prove c.
Looks to database to see if there's c(X)
and c is 3.
Since 3 is not 1,
it fails that
and backtracks to b
and tries to find another solution for b
so it goes to rule 2 again
b(X), trying to find a new value for d.
It finds fact 5, d(2),
instantiates X to 2 there
and e to 2,
then looks in the database to find if there's an e(2).
Fact 7 is e(2)
so it then instantiates X to 2
and er in rule 1,
then establishes c(X) to c(2)
looks in the database to see if there is c(2),
it's not there,
so backtracks again,
tries to find an alternative solution to b
uhm ... there's no alternative solution to b
because there are no alternative solutions to d and e,
so ultimately it ah ...
so then ... then it goes on to rule number 3,
tries rule number 3 for b
and finds that b(X) can be proven
if f(X) can be proven,
so it looks in the database for f(X),
finds fact 8, f(3)
so X is instantiated to 3,
hum, back then ...
goes back to the first rule, b(X) and c(X)
the X there is instantiated to 3,
looks in the database to see if there's a 3,
fact number 9 says c(3) so that succeeds
and so the subgoals b and c have succeeded
therefore the principle goal a(X) has succeeded
and X is instantiated to 3.

MSc. COOMBS AND STELL BACKTRACKING EXERCISE (CUT): SUBJECT 12
(The only subject to do cut version before original)

1 Well it's trying to prove the theorem a(X)
2 and has a rule a(X) if b(X) and c(X)
3 so in order to prove a(X) it has
4 first to prove the subgoals b(X) and c(X)
5 There's no b(X) in the database
6 therefore it has to look to another rule
7 and there is a second rule b(X) if d(X), cut, e(X)
8 So then has to prove first of all d(X)
9 in the database there is a d(1)
10 so it has proven 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 It's now got a proof for b(X)
23 it's now got to prove c(X) ...

E: What's the present instantiation of X?

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).
30 Having proven b(X) it goes on to prove c(X)
31 so we have one solution for b(X) which is b(1).
32 Going on then to prove c(X)
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 and again 4 is not 1 so that fails.
39 So a(X) fails.
Well it would go down
look at its database which is this
and find and the first instance of a(X)
and then uhh see if it could satisfy it
Uhm ... so if it's a rule it would look ...
if it wasn't a rule then it would just see
if it matched it.
If it is a rule then it looks to see if it can satisfy
the clauses in the rule
and it would go from left to right through the rule
and find b(X)
and it would go on down the database trying to find b(X)
and that's on that's in this goal queue thing.
It would put a(X) in the queue and find b.
So b(X) is being examined, and it will go down
look at b(X) and then again if ...
Uhm it would take the first instance of b(X)
and then look to see if it could satisfy
uhm b(X) which means it would pick up d.
D is the first clause in the rule, and then go off again
So we've got a(X) and b(X) on these ...
1 and 2 are on the goal queue.
It would go down and find ...
when it came across the first instance of d it would uhh
see if it could match it against d(X),
it would match it
so ... because X is a variable
and there's only one variable d
and there's one constant in this instance of d
so X is instantiated to ... well it's not actually,
it's called ... this is where I ...
I don't know if it would actually be attached to 1
or if 1 and X would share _1 or _5 or something ...

E: No. X will now pick up that value so ... X equals 1 now ...

So X is equals 1.
Then it would uhh that's right, come out of here,
go back here,
and then go on to the next,
try and satisfy b by looking at e uhh
and go down the database until it found e
and the same thing would occur
X would take the value ... become
X would take the value of 1?
X and 1 would be the same thing
Uhm ... and then it would evaluate that ...
Ahm ... so you've ... hmmm.
You've got d(1) and e(1) ... if d(1) and e(1)
then you can say b - oh yes - I think you have to -
X in this case has to be the same thing,
so that when you got to d(1)
X would be equal to 1, e would be equal to 1,
then it would go off and see if there was an e ...
which it finds, comes back then ...
53 X here as well would be instantiated to 1
54 and that will have been satisfied and so goes off ...
55 is no longer in the goal queue whatever
56 so then you come back to the other rule
57 which is waiting - er the other ...
58 yes rule, or goal now it is I suppose,
59 rule which is a goal waiting to
60 be satisfied and X is the value 1,
61 so all these will be given the value of 1
62 and try and find c with X of the value of 1
63 in the database.
64 So it goes down and it looks
65 and it doesn't find it.
66 So it goes back again, and says,
67 I haven't got it so X can't equal 1.
68 Maybe it's er ...
69 So this rule of b has to be ...
70 The first rule of b doesn't satisfy this
71 rule so you have to, go back and X becomes X
72 I'd have thought.
73 Just a minute ...
74 The thing that worries me is that we've got all
75 these 1's dangling about all over the place,
76 But I'm ... when it comes back up again and
77 says Ok X is equal to 1 then it's got all
78 these things tagged as 1, so when it says no
79 that isn't right, it just overwrites does it?

E: Not yet.

80 It doesn't. So they're all still tagged

E: They're all still 1 at the moment. 
What's the next action it takes?

81 The next action it takes is to go off
82 and look for another rule for b.
83 Yeah ... uhm ... b was X as itself ... well b
84 uhm ... I'm not quite sure when the change
85 in the variable comes about but I'd ...
86 but b came ... uhhmm
87 What does it do now?
88 Well my ... what I'd say is it goes down
89 and looks for another instance of b in the database,
90 b of something or other
91 and ... if it's able to match b of ahhhhh!
92 with b(X) with b(X),
93 but I think I've jumped a stage,
94 but anyway it finds that,
95 and says well if I can satisfy this b
96 then I can carry on and satisfy a. Uhm.
97 And in order to satisfy this b I have to find f(X)
98 Is that in the database in some form
99 that can be matched to that.
100 And it finds that f(3), X becomes 3,
101 uhm and I think that's where it ...
102 All right so f ... X in this rule becomes 3
103 and b goes back with a value of X being three
104 and all the X's in here are matched to 3.
105 So then it tries to find, it says ok
106 see if any clauses in a can be matched in the database.
107 So c(x) ... no just a minute.
108 Yeah the other clause is ...
109 Well it goes on to see if the rest of the clauses
110 on which this is dependent can be matched.
111 So c(3) is that in the database.
112 So it looks down and it find c(3) yes that's ok,
113 Ahm ... and then yes that's ok, so a ... X ... a can be
114 satisfied when X has the value of 3.

MSC. COMBS AND STEILL BACKTRACKING EXERCISE (CUT): SUBJECT 13

1 Right, so I'm trying to find out if a(X) is true
2 I try the first goal. er ... rule or fact in the database
3 uhm with predicate a,
4 uhm X is a variable I presume,
5 so a can be satisfied,
6 a(X) can be satisfied if b,
7 b(X) is true -
8 it remembers that
   [indic. a(X) in head of Clause 1]
9 goes off looking down the list
10 to see if it can find b(X)
11 any predicate, any rule or fact
12 with the predicate b, really -
13 it doesn't matter what it's got as its arguments -
14 oh it has to have one argument
15 and um find this one
   [indic. Clause 2],
16 it comes to the first one and it says if ...
17 b is true, b(X) is true
18 if d(X) is true,
19 so it goes off to look for a predicate
20 with d and one argument
21 and it finds d(1),
22 X is instantiated to 1 ...
23 then it comes back -
24 is that right?
25 Yeah, X is instantiated to 1 all through up here
   [indic. Clause 2]
26 and here
   [indic. body of Clause 1]
27 and here
   [indic. head of Clause 1]
28 and then um it gets
29 so then it'll say ok
30 I can go onto the next clause
31 and the next clause is the cut,
32 which means that once you've passed the cut,
33 you can never go back,
34 so it keeps going, um ...
35 I'm - fascinates me this - no moral decisions
36 to be made or anything like that
37 just kind of you know, no kind of mucking about,
you just go straight through!
So e(X) is — oh I mean the next one to be satisfied
and X is now stuck on 1,
you can't you can't change its value.
So you need to try and find e —
oh yeah that's — yeah — back all these,
X, X, X, and that one are 1
[indic. e(X)]
when that's found
and once this is passed it's stuck like that
so you've got to ...
it goes off and tries to find e(1)
and it finds it,
and it says oh, that succeeds
so that clause can succeed and that b succeeds
[indic. Clause 2]
and it goes back to a
and says yes, b is is true
and in order for a(1) to be true
c(1) has to be true too,
so it goes off to look for c(1) —
can't find it.
So it goes, fail,
goes out ... How does it fail?
Goes out of c, goes back to b ...
I don't know if ... does it? Yeah,
but I think it kind of goes back here
[indic. e(X)]
goes kind of in the other direction,
comes in the back end of d
and it says I can't —
I can't deinstance e
so I can look for another — pause —
What does it do then?
I know it can't go past here
[indic. !]
but I think that ... I don't think ...
it has to be e(1)
because it already is 1,
so it says
so basically you can't do anything.
I'm not sure about the actual minutiae
of what's going on here
[indic. e(X)]
but I know it says you can't go back that way.
In other words b fails, and a fails because b fails.

E: And what about this clause here?
[indic. Clause 3]
It never gets to it ...
APPENDIX 6

Transcripts of Alex's Parsing Programs

Original Bicycle Program

basicpart(rim).
basicpart(rearframe).
basicpart(gears).
basicpart(nut).
basicpart(spoke).
basicpart(handles).
basicpart(bolt).
basicpart(fork).

assembly(bike, [quant(wheel,2), quant(frame,1)]).
assembly(wheel, [quant(spoke,20), quant(rim,1), quant(hub,1)]).
assembly(frame, [quant(rearframe,1), quant(frontframe,1)]).
assembly(hub, [quant(gears,1), quant(axle,1)]).
assembly(axle, [quant(bolt,1), quant(nut,2)]).

partsof(X,[X]) :- basicpart(X).
partsof(X,P) :- assembly(X, Subparts),

partsoflist(Subparts, P).

partsoflist([],[]).
partsoflist([quant(X, )], [Tail], Total) :-

partsof(X, Headparts),
partsoflist(Tail, Tailparts),
append(Headparts, Tailparts, Total).

append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).


Alex's Parser: Version 1

basicpart(d).
basicpart(n).
basicpart(v).
basicpart(a).
assembly(np,[d,n]).
assembly(vp,[a,v]).
assembly(np,[n]).
assembly(vp,[v]).
assembly(s,[np,vp]).
lexitem(d,the).
lexitem(n,man).
lexitem(n,woman).
lexitem(n,anthony).
lexitem(a,will).
lexitem(v,hit).

Alex's Parser: Version 2:
basicpart(fred).
basicpart(man).
basicpart(the).
basicpart(hit).
basicpart(woman).
basicpart(will).

assembly(s,[np, vp]).
assembly(np,[d, n]).
assembly(vp,[a, v, np]).
assembly(n,[fred]).
assembly(v,[hit]).
assembly(d,[the]).
assembly(a,[will]).
partsof(X,[X]) :- basicpart(X).
partsof(X,P) :-
  assembly(X, Subparts),
  partsoflist(Subparts, P).

partsoflist([],[]).
partsoflist([Head | Tail], Total) :-
  partsof(Head, Headparts),
  partsoflist(Tail, Tailparts),
  append(Headparts, Tailparts, Total).

append([], L, L).
append([X | L1], L2, [X | L3]) :-
  append(L1, L2, L3).


KOWALSKI, R. (1973). 'Predicate logic as programming language', Memo No. 70, Department of Computational Logic, School of AI, University of Edinburgh.


