

Open Research Online

The Open University's repository of research publications and other research outputs

An application of formal semantics to student modelling : an investigation in the domain of teaching Prolog

Thesis

How to cite:

Fung, Pat (1989). An application of formal semantics to student modelling : an investigation in the domain of teaching Prolog. PhD thesis The Open University.

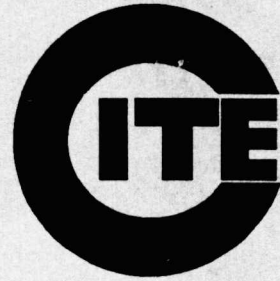
For guidance on citations see [FAQs](#).

© 1989 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.

oro.open.ac.uk



**Centre for Information
Technology in Education**

**An Application of Formal Semantics to Student
Modelling:**
an investigation in the domain of teaching Prolog
Pat Fung

CITE Ph.D. Theses No. 5



**Institute of Educational Technology
Open University
Walton Hall
Milton Keynes
Great Britain
MK7 6AA**

31 0005424 5



DX88784

UNRESTRICTED

An Application of Formal Semantics to Student Modelling:

an investigation in the domain of teaching
Prolog

Pat Fung, B.A., MSc.

**Thesis submitted in partial fulfilment of requirements for the degree of
Doctor of Philosophy in Cognitive Science
Open University, Milton Keynes,**

October 1989

Author's Number: 11702382X
Date of Submission: 29th September 1989
Date of Award: 18th December 1989

HIGHER DEGREES OFFICE
LIBRARY AUTHORISATION FORMSTUDENT: PAT FUNG SERIAL NO: _____DEGREE: PH. DTITLE OF THESIS: AN APPLICATION OF FORMAL SEMANTICS
TO STUDENT MODELLING: AN INVESTIGATION
IN THE DOMAIN OF TEACHING PROLOG

I confirm that I am willing that my thesis be made available to readers and maybe photocopied, subject to the discretion of the Librarian.

SIGNED: P. Fung DATE: 25.9.83

Abstract

This thesis reports on research undertaken in an exploration of the use of formal semantics for student modelling in intelligent tutoring systems. The domain chosen was that of tutoring programming languages and within that domain Prolog was selected to be the target language for this exploration. The problem considered is one of how to analyse students' errors at a level which allows diagnosis to be more flexible and meaningful than is possible with the 'mal-rules' and 'bug-catalogue' approach of existing systems. The ideas put forward by Robin Milner [1980] in his Calculus of Communicating Systems (CCS) form the basis of the formalism which is proposed as a solution to this problem. Based on the findings of an empirical investigation, novices' misconceptions of control flow in Prolog was defined as a suitable area in which to explore the application of this solution. A selection of Prolog programs used in that investigation was formally described in terms of CCS. These formal descriptions were used by a production rule system to generate a number of the incomplete or faulty models of Prolog execution which were identified in the first empirical study. In a second empirical study, a machine-analysis tool, designed to be part of a diagnostic tutoring module, used these models to diagnose students' misconceptions of Prolog control flow. This initial application of CCS to student modelling showed that the models of Prolog execution generated by the system could be used successfully to detect students' misunderstandings. Results from the research reported here indicate that the use of formal semantics to model programming languages has a useful contribution to make to the task of student modelling.

Dedication

To the memory of our brother John, who died in March 1987

Acknowledgements

Thank you to my supervisor, Dr. Mark Elsom-Cook, who has been a constant source of encouragement throughout my doctoral studies. He always, without fail, made time to discuss and criticise my work. On the other hand, at those times when the last thing on earth I wanted to do was to discuss my work, he showed patience and understanding. What more could a research student want of a Ph.D. supervisor?

I would like to thank Ann Jones, not only for her willingness to read and criticise my work, but also for the moral support she has given me in times of crises throughout my research studies.

Thank you to all the research members of CITE, past and present, whose help, encouragement and friendship have been invaluable to me. In particular I would like to thank the following: Tim O'Shea, Eileen Scanlon, Di Laurillard, Claire O' Malley, Alistair Edwards, Rick Evertz, Sara Hennessy, Mike Baker, Simon Holland, Kate Stainton-Ellis, Rachel Hewson and Fiona Spensley (a special thank you to Fiona for the final proof reading of my thesis).

For very much the same reasons, I would like to thank the following research members of HCRL, Marc Eisenstadt, Mike Brayshaw, Tony Hasemer, Tim Rajan and John Domingue.

Thank you to the members of the Prolog groups at Edinburgh and at Sussex who have read and criticised my papers and shown interest in my work.

Thank you to all the D309 summer school students who contributed to the empirical work in this thesis. A thank you also to all the summer school organisers and tutors who directly or indirectly were a help to me in doing that work. In particular my thanks to Ingrid Slack and Hank Kahney.

A special thank you to Olwyn Wilson, who has always been a source of help and common sense in times of need and to Di Mason and Pat Cross for their cheerfulness in the face of my frequent interruptions to their work.

I owe a particular thank you to Dr. Benedict Du Boulay. It was he who initially gave me the opportunity to discover the excitement of computers and Artificial Intelligence. I am in debt to him for that and for the continuing interest he has shown in my work during my Ph.D studies.

Finally, I owe the biggest debt of thanks to my family. To Igie, without whose love and support this work would not have been possible. To Tony and Jackie, for being understanding about a mother who is rarely home. To sisters and brother, Cathy, Sheila, Val and Quentin for their love and encouragement when the world seemed tough. To my mother, who really is the best mother in the world.

This work was funded by an ESRC grant (no. 000426624233)

Table of Contents

Introduction	1
Chapter One	
1. Related research	6
1.1. What sort of difficulties do novice programmers have?	6
1.1.1. Finding an algorithm	8
1.1.2. Program planning - using plans	10
1.1.3. Problems in program coding	13
1.1.4. Debugging problems	15
1.1.5. Difficulties related to learning Prolog	17
1.2. What help is available for Prolog novices?	23
1.2.1. Prolog courses	24
1.2.2. Prolog environments	30
1.2.3. Summary	43
1.3. Tutoring systems	46
1.3.1. Lisp tutors	48
1.3.2. Pascal Tutors	55
1.3.3. Help in Fortran	59
1.4. Tutoring for Prolog	60
1.4.1. Tutoring modules - PITS	61
1.5. The Prolog Interpreter	61
1.6. Summary	62
Chapter Two	
2. Novices' misconceptions of the Prolog interpreter	65
2.1. A proposed taxonomy	66
2.1.1. Errors related to Prolog search and backtracking	68
2.1.2. Incomplete or faulty knowledge of unification.	72
2.1.3. Confusion over the effect of the cut operator	74
2.1.4. Implications	76
2.2. An empirical study	78
2.2.1. The experiment	79
2.2.2. Problem design	82
2.2.3. Results	89
2.2.4. Comments	108
2.3. Summary	110
Chapter Three	
3. Formal Models	112
3.1. Modelling students	112
3.2. Formal descriptions	116
3.3. A Calculus of Communicating Systems	118
3.3.1. Synchronised communication	119
3.3.2. Observational equivalence	121
3.3.3. Composition	125
3.3.4. Applying the expansion theorem	126
3.3.5. Restriction	128
3.3.6. Equivalences	130
3.4. Summary	134

Chapter Four

4. A Prolog Application	135
4.1. System overview	136
4.2. Producing formal descriptions of Prolog programs	138
4.2.1. Query components	140
4.2.2. The condition component	143
4.2.3. The right hand side component	144
4.2.4. Fact and nomore components	148
4.2.5. A program converted	148
4.3. Expanding the formal description of a program	150
4.3.1. Composing two program components	151
4.3.2. Applying the expansion theorem	152
4.3.3. Tagging the silent communications of a program	155
4.3.4. Restricted observation of a program	158
4.4. Summary	160

Chapter Five

5. Production rule modelling	162
5.1. A production rule approach	162
5.2. Developing rule conditions	166
5.2.1. Node-types	167
5.2.2. Node positions	172
5.3. Rule actions	174
5.3.1. Action at nochoice-nodes	175
5.3.2. Action at disjunctrhs and disjunctprime nodes	177
5.3.3. Actions at disjunct nodes	177
5.3.4. Action at failchoice-nodes	181
5.3.5. Action at final nodes	182
5.4. A ruleset for normal Prolog search	183
5.5. Modelling misconceptions - some examples	186
5.5.1. Try once and pass	186
5.5.2. Redo from left	187
5.5.3. Facts before rules	191
5.5.4. One pointer per clause	194
5.6. Summary	197

Chapter Six

6. An evaluation.	199
6.1. A second empirical study	200
6.1.1. Subjects taking part	200
6.1.2. Experiment design	201
6.1.3. Problem design	204
6.1.4. Confidence ratings	205
6.1.5. Data analysis	206
6.1.6. Machine-analysis	206

6.2. Experiment results	209
6.2.1. Answers analysed	209
6.2.2. Errors found	210
6.2.3. Differences between analyses	212
6.2.4. Confidence rating results	219
6.2.5. Discussion of results	223
6.2.6. Machine-analysis (1987 summer school experiment)	225
6.3. Machine-analysis comments	230
6.4. Summary	232
Chapter Seven	
7. Conclusions	236
7.1. Achievements	236
7.1.1. An initial taxonomy	237
7.1.2. A study of novices' models of the Prolog interpreter	238
7.1.3. The development of a machine-analysis tool	239
7.1.4. A production rule description language	240
7.1.5. A computer-based empirical study.	241
7.2. The significance for intelligent tutoring	243
7.2.1. A contribution to diagnosis in tutoring systems	243
7.2.2. A contribution to student modelling	245
7.2.3. A contribution to empirical research	245
7.2.4. Summary	246
7.3. Future directions	246
7.3.1. Extensions to existing work	247
7.3.2. Longer term research - the role of formal semantics	251
References	254
Appendices	
A. Empirical study 1987	
A1. Experiment booklet	264
A2. Table of results	272
A3. Individual results	274
A4. Percentages of problems showing errors	282
B. Listing of programs	
B1. Conversion of Prolog programs	284
B2. Expansion theorem	292
B3. Production rule system	300
Appendix C. Empirical study 1988	
C1. Screen dumps of programs used	310
C2. Results of machine-analysis	316
C3. Results of hand-analysis	320
C4. Confidence ratings	322
C5. Machine-analysis of 1987 results	326
D. Screen dumps of CCS representations of programs	330

Table of figures

Chapter One

Fig.1.1	Areas of skill used in constructing a program	7
Fig.1.2	Byrd Box Model of simple Prolog goal.	26
Fig.1.3	Model of embedded goals.	27
Fig.1.4	An Arrow Model of Prolog control	28
Fig.1.5	A Tree Model	29
Fig.1.6	Sample standard Byrd box trace	35
Fig.1.7	Sample trace of Prolog Trace Package	38
Fig.1.8	Snapshot of APT tracing tool	40
Fig.1.9	TPM trace of a program at the point of goal satisfaction	42
Fig.1.10	Detail of TPM status box	42
Fig.1.11	Long distance view of TPM trace	43
Fig.1.12	Approximation of teaching strategy of Greaterp	51
Fig.1.13	Design of Impart tutoring system	52
Fig.1.14	Section of Trill's semantic network	54
Fig.1.15	Matching process in Proust	57

Chapter Two

Fig.2.1	Arrow diagram	68
Fig.2.2	Control flow snapshot	68
Fig.2.3	Facts before rules	69
Fig.2.4	Rules before facts	70
Fig.2.5	Correct interpretation	71
Fig.2.6	Incorrect interpretation	71
Fig.2.7	Redo from left	72
Fig.2.8	Can two variables have the same value?	73
Fig.2.9	The correct answer	73
Fig.2.10	Correct interpretation of the scope of the cut operator	74
Fig.2.11	Correct scoping of the cut operator	75
Fig.2.12	Under-estimation of the scope of the cut operator	75
Fig.2.13	Correct scoping of the cut operator	76
Fig.2.14	Over-estimation of the scope of the cut operator	76
Fig.2.15	Programming experience prior to the O. U.course.	79
Fig.2.16	Correct interpretation	83
Fig.2.17	Incorrect interpretation	84
Fig.2.18	Problem design	88
Fig.2.19	Breakdown of total number of errors (217).	91
Fig.2.20	Number of errors made by each student	91
Fig.2.21	Distribution of numbers of errors made by students	92
Fig.2.22	Breakdown of identified errors	93
Fig.2.23	Distribution of 'one pointer per clause' errors	97
Fig.2.24	Distribution of the 'meta-knowledge' misconception	98
Fig.2.25	Number of students related to error type	99
Fig.2.26	Problems and expected errors	101
Fig.2.27	Distribution of errors across problems	102
Fig.2.28	'Redo from left' misconception	103
Fig.2.29	'One pointer per clause' misconception	103
Fig.2.30	Comparison of errors	105
Fig.2.31	Errors related to programming experience	106
Fig.2.32	Comparison of error averages	107

Chapter Three

Fig.3.1	Actions of two systems	119
Fig.3.2	Two agents combined	120
Fig.3.3	A more abstract representation	121
Fig.3.4	Equivalent behaviours	122
Fig.3.5	A nondeterministic choice of actions - Fred	123
Fig.3.6	A nondeterministic choice of actions - Hatter	124
Fig.3.7	Agent f	124
Fig.3.8	Agent h	124
Fig.3.9	Composition of agents f and h	125
Fig.3.10	Definition of composition	126
Fig.3.11	Composite machine $f h$ expanded	127
Fig.3.12	Definition of expansion	128
Fig.3.13	Machine $f h$	129
Fig.3.14	Restricted machine $f h \setminus a$	129
Fig.3.15	A formal definition of restriction	130
Fig.3.16	Equivalence levels	130
Fig.3.17	A binary relation over f	131
Fig.3.18	A silent communication action	132
Fig.3.19	Observationally equivalent	132
Fig.3.20	Observational equivalence of customers behaviours.	132
Fig.3.21	Examples of observational equivalence	133

Chapter Four

Fig.4.1	Process outline	137
Fig.4.2	Formulating components	139
Fig.4.3	Components of program	139
Fig.4.4	Actions of query component	142
Fig.4.5	Actions of condition component	144
Fig.4.6	Actions of the right hand side	145
Fig.4.7	Actions of fact and nomore components	148
Fig.4.8	Formal description of actions of machines	150
Fig.4.9	Actions of fact component and nomore component	151
Fig.4.10	Composition of fact and nomore machines	152
Fig.4.11	Possible sequences of actions from the expansion of the	153
Fig.4.12	Possible complementary actions of SA1- and SA1	155
Fig.4.13	Restricted expansion of (SA1(SA1-)) (SA2(FA2-))	159
Fig.4.14	CCS tree representation of program 'p if a'. 'a'.	161

Chapter Five

Fig.5.1	CCS tree of p if a & b.	163
Fig.5.2	Section of semantic tree showing branches traversed	163
Fig.5.3	Branches of CCS tree traversed in 'try once and pass'	164
Fig.5.4	Components of production rule interpreter	166
Fig.5.5	Disjunct points with same names	168
Fig.5.6	Disjunct point with different names	168
Fig.5.7	Another 'different name' disjunct	170
Fig.5.8	No-choice nodes in CCS tree	172
Fig.5.9	Directions of actions	175
Fig.5.10	Ruleset for normal Prolog search	183
Fig.5.11	Returning to disjunctrhs	190
Fig.5.12	Identifying a rule clause	192
Fig.5.13	Once used, other instances of the branch are lopped off.	195

Chapter Six

Fig.6.1	Layout of window shown to student	203
Fig.6.2	Window with student input	203
Fig.6.3	Total number of answers analysed successfully	209
Fig.6.4	Subject by subject breakdown of answers	210
Fig.6.5	Breakdown of 190 errors found by machine-analysis	211
Fig.6.6	Breakdown of 184 errors found by hand-analysis	211
Fig.6.7	Comparison of error totals	213
Fig.6.8	Hand-analysis results of subject 15.	214
Fig.6.9	Prediction given by subject 15 for problem one	214
Fig.6.10	Layout of buttons for goal 'p'.	215
Fig.6.11	Prediction for problem one given by subject 9.	216
Fig.6.12	'Abbreviated' answers of subject 28	217
Fig.6.13	Prediction given by subject 16	218
Fig.6.14.	Breakdown of total number of predictions	221
Fig.6.15	Relationship between confidence and accuracy	221
Fig.6.16	Some individual cases showing differences	222
Fig.6.17	Total number of errors, hand-analysis, 1987 data.	225
Fig.6.18	Breakdown of identified errors, hand-analysis, 1987.	226
Fig.6.19	Differences, hand/machine-analysis, 1987 data	227
Fig.6.20	Hand-analysis results for subject 29, 1987 data	228
Fig.6.21	Results of hand-analysis of subjects 6 and 8, 1987 data	228
Fig.6.22	Results of subject 5, hand-analysis, 1987 data.	229

Chapter Seven

Fig.7.1	A first representation of a variable	248
Fig.7.2	Components of a Var machine	248

An Application of Formal Semantics to Student Modelling: an investigation in the domain of teaching Prolog

Introduction

The research reported in this thesis was undertaken in the context of developing a diagnostic tutoring module which would be capable of making a useful contribution to an intelligent tutoring system in the domain of programming languages. The problem addressed is one of student modelling, since the ability to provide meaningful on-line diagnostic help to a novice programmer centres on the successful identification of that student's faulty or incomplete perception of the programming language. To identify that a student has made an error is rarely in itself sufficient, it is far more important to identify the most likely misunderstanding which has led to the error, i.e. to identify the faulty or incomplete model of the language which the student may have formed. A possible solution to the problem of identifying such models, put forward in this thesis, is the use of a formal semantics. A very exact picture of program behaviours can be obtained by formally describing the programming language being tutored. This information can then be utilised to construct the models which novices form and can provide the basis for appropriate diagnostic help where this is necessary. The formalism which has been experimented with in applying this solution to the problem of student modelling is based on R.Milner's [Milner 1980] Calculus of Communicating Systems (CCS). The programming language Prolog was chosen as the vehicle to investigate its use. In outline, the objectives in exploring this solution were as follows. An initial goal was to define an area in the task of learning the programming language which presents difficulties for novice Prolog

programmers. The next objective was to investigate the ideas of CCS as a formalism which could provide diagnostic help in representing students' models of that area. Subsequently, these ideas were translated into a Prolog context and incorporated in the initial development of a diagnostic component of a tutoring system. This work was then evaluated and the advantages and disadvantages of this approach to the problem of student modelling considered in the light of the evaluation. The following outline of the chapters of this thesis indicates the structure of the research undertaken in the course of pursuing these objectives. Essentially this entailed defining a specific area of difficulty for novice Prolog programmers, based on an overview of the existing research literature and the results of an empirical study. The formalism proposed for constructing models of novice programmers' misconceptions was then investigated and its application to this area explored. A program designed to illustrate the use of this formalism in diagnosing Prolog novices' misunderstandings was implemented and subsequently evaluated in a second empirical study.

In chapter one we put the current work into the context of previous research in this area. To do this we initially consider what research has shown us of the difficulties which novice programmers experience when they begin to learn a programming language, since it is these difficulties which provide the motivation for this work. Drawing on the research literature in this field, a significant conclusion emerges. In addition to many of the problems generally encountered by students learning to program, students learning Prolog face additional problems which are specific to that language and these are considered. One such problem in particular is the task of understanding the behaviour of the Prolog interpreter.

Having considered the problems which research has shown that students encounter, a brief survey is made of the help which is available to novice programmers to overcome these problems. Research in this area has provided automated help which ranges from trace-packages and tools to tutoring systems, but in this overview we indicate certain shortcomings in this help. We regard these shortcomings as evidence of the need for intelligent on-line tutoring. An essential feature of such tutoring must be that it addresses the problem of student modelling in a meaningful way. It is in this context that we advocate the development of a tutoring module for diagnosing those errors which are indicative of novices' faulty perceptions of the programming language.

In chapter two we draw on our review of the problems encountered by students learning to program in order to define an area of difficulty for Prolog novices, the task of understanding the actions of the Prolog interpreter. In chapter one we described certain models of the interpreter which are offered to novices to help them interpret the execution behaviour of Prolog, but students often form their own incomplete or faulty models. In this chapter we discuss possible misconceptions of the Prolog interpreter which may underlie these incorrect models. Following this discussion we describe an empirical study in which students were asked to predict the steps taken by the interpreter in proving certain Prolog programs. This study was undertaken to investigate in more detail the models students form of the backtracking process in Prolog. The results confirmed that Prolog's backtracking processes do present difficulties for novices and that models of the interpreter formed by students in the early stages of learning the language were indeed often faulty and incomplete. Some

of the models which students had formed of Prolog backtracking were easier to identify than others and the results showed that particular models appeared relatively consistently in their answers. In view of the results, a selection of these faulty models were used as the basis for exploring the use of CCS in constructing computational models of the misconceptions of novice Prolog programmers. In the third chapter we explain why this formalism was chosen as the basis for representing such models and discuss the ideas central to CCS, i.e. synchronised communication and the notion of observational equivalence, since it is the potential of these ideas which make CCS of particular interest in relation to the tutoring of programming languages.

Translating the ideas of CCS to Prolog was an important stage in the research project and in chapter four we look at the process of developing a formal description of a Prolog program. Using a simple program as an example, we then show how this formal description provides the terms from which we derive the possible backtracking behaviours, correct and incorrect, of that program. It is this semantic information, generated from the formal description, which is used in constructing models representing students' misconceptions and which can be used for diagnostic tutoring. Chapter five outlines the design of the system in which this information is incorporated and discusses the development of the production rules which are used in the modelling process. Using a production rule interpreter approach, the basic set of rules which model the normal search process in Prolog can be modified to reproduce the faulty models of Prolog search which novices form. The primitives used in the ruleset form the potential building blocks which can be used to construct the particular model of backtracking that a student may have developed.

Chapter six is concerned with a second empirical study which was undertaken in order to provide a means of evaluating the work of representing students' models of Prolog execution. This was a computer-based experiment investigating students' models of backtracking and the results were machine-analysed using the computational models and subsequently hand-analysed. The results of the earlier experiment discussed in chapter two were also retrospectively machine-analysed. The relative merits of the analyses are discussed and implications of the results for the system being developed are considered in the light of these results.

The thesis concludes, in chapter seven, by summarising the results of the research undertaken in relation to its goals, looking both at what has been achieved in the course of this work and what appear to be the most promising avenues for further investigation. The use of a formal semantics in generating the computational representations of students' models of Prolog backtracking proved to be a successful step in identifying misconceptions of the Prolog interpreter. It also indicated further potential in this approach that has not yet been explored, but which could ultimately contribute to solving other problematic aspects of student modelling which were not the focus of this thesis.

Chapter One

1. Related research

In this chapter we give a brief overview of research related to this thesis. The purpose in doing so is to answer two questions, "What sort of difficulties do novice programmers have, particularly Prolog novices?" and "What help is available to assist the latter in overcoming these difficulties?" Both questions are essential in identifying an area of difficulty for novice Prolog programmers and in putting the current work of developing a diagnostic tutoring module into context in the field of tutoring programming languages. We therefore look at problems which novices encounter in learning to program, as reflected in the research literature and end this overview by considering the relevance of this to Prolog novices. Having outlined the areas in which novices experience problems, we then, in the concluding section of this chapter, discuss the extent to which research to date has attempted to alleviate the difficulties of Prolog novices by the development of programming tools and tutoring systems.

1.1. What sort of difficulties do novice programmers have?

The term 'novice' has been used in various studies of programmers [Taylor 1987], [Kahney 1982] and is generally taken to mean a subject who has had little or no experience of programming. Although [Shneiderman 1976] used the expression rather more precisely to denote the level of programming experience, i.e. one who has attended or

completed a first programming course, the term will be used here in a more general sense covering both categories (this will be discussed further on p.17-18). Many of the studies referred to here have been of novices learning languages other than Prolog. This in no way invalidates their relevance to this work, but rather helps to underline the commonality of some difficulties across languages whilst helping to identify those problems specific to Prolog.

One must also be aware of the warning given in [Sheil 1981] that the complexity of programming behaviour makes evaluation difficult. It is not possible to say with complete certainty to what extent problems in one area of the skills involved in programming are due to or are the cause of, problems in another area. Any classification of novices' problems is therefore to some extent an arbitrary one. However, for the purposes of this thesis, we classify the following four 'areas' as ones in which novices are known to experience problems, finding computational solutions to problems, program planning, program coding and debugging programs.

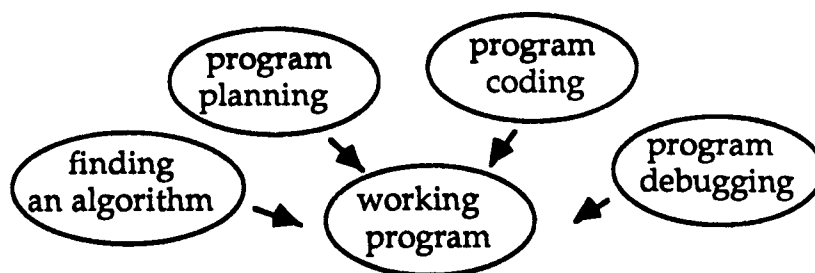


Fig.1.1. Areas of skill used in constructing a program

We see these areas as relating approximately to the steps of finding an algorithm for the solution of the programming problem, planning a computer program expressing this solution, coding and running the

program and finally the process of refining or debugging the program. The difficulties which novices experience in these areas will be discussed in turn in the following sections.

1.1.1. Finding an algorithm

The problems which novice programmers are asked to solve are not, or should not be intrinsically complex [du Boulay & O'Shea 1981], often being related to background knowledge which they would be expected to possess. This is not always true, particularly if the student is studying a programming language in isolated circumstances, but holds true in most cases. Most often the problems given to novices are such that they would be relatively easy to solve in everyday life, such as finding the largest number in a given set of numbers, finding how many elements there are in a given list, reversing the elements of a list, or solving well known puzzles. Given paper and pencil and allowed time for trial and error, most students would soon find an algorithm for arriving at a correct solution. The every-day natural language algorithms which students might put forward as solutions to such problems are not however necessarily suitable ones for representing computationally. The task of formalising their solution into an algorithm that can eventually be expressed in a computer programming language is a new experience.

Given, of course, that there may be significant differences between individuals which make it much harder or easier for some to accomplish this task than for others [Kahney 1982], this process of finding an algorithm is nevertheless often a difficult one for novices. In his study, Kahney [1982] points to the amount of knowledge that novices must employ in this process of describing the problem and its solution

in a way that can be utilised as a computer program. Students must not only know the 'everyday' solution to the problem. They must also have enough understanding of the particular programming language to relate features of the problem to the programming constructs which would be pertinent to the solution.

As stressed by [Jones 1981], in order to construct a valid representation of the problem, the novice needs both a clear conceptual model of the problem and an awareness of the programming language constructs. In this context a clear conceptual model of the problem would imply one which contained the problem's salient points. It would imply the ability of the student to discard any data irrelevant or superfluous to finding an algorithm for its solution. An awareness of the programming language would imply familiarity with the basic structures of the language and a knowledge of how to manipulate those structures. It is highly likely that a novice, not having such an awareness, would form a model of the problem and its solution drawn only from her own everyday knowledge.

It has been shown that novices do not come to programming as 'empty heads' [Kahney 1982 pp 1-3], but bring to programming a background of existing knowledge to which they attempt to relate their 'new' knowledge. This 'active' learning in a domain where students may not be aware of which items of their existing knowledge-store are relevant and which are not, can lead to inappropriate expectations [Jones 1984]. A model of the problem and its solution which is too firmly rooted in everyday knowledge can be a hindrance, since it may obscure the goal of finding a solution that is amenable to computation.

A study undertaken to clarify the issues governing suitability of programming languages for programming tasks [Petre & Winder 1988] revealed an interesting insight into the importance which 'expert' programmers attach to this stage of the programming process.

Protocols showed that they considered that the task of finding an algorithm was far more important than the actual coding of that algorithm. However, the data revealed that the algorithms referred to by these experts were in most cases written in 'pseudo-code', a mixture of natural language and programming language terms (in some cases from more than one programming language). This indicates to what extent they brought their awareness of programming constructs to the task of finding a suitable algorithm, so it is not surprising that novices, with relatively little awareness, face difficulties in this area.

In summary then, work on novice programmers and their approaches to finding algorithms to solve given programming problems has shown that they are likely to have difficulties in representing the solutions to those problems in a computationally viable way. Suggested sources of these difficulties have been the misapplication of 'everyday' problem-solving techniques, inadequate knowledge of appropriate programming constructs and the sometimes misleading similarities between the semantics of natural language and of programming languages.

1.1.2. Program planning - using plans

The stage of the programming process being discussed in this section refers to the phase in which the algorithm is turned into a program structure or outline (it is perhaps worth mentioning again the arbitrary nature of dividing the programming task into stages [see section 1.1],

which is used in this case to provide a framework for discussing the problems novices encounter). The programmer is not writing the final code, but is planning the shape of the program which will implement the algorithm. At this point of planning the programmer draws on her knowledge of the structures of the language being used and formulates an overall plan of how to implement the algorithm. The achievement of this main plan will normally entail the use of lower level plans.

The role of plans in programming has been the subject of much recent investigation. The concept of programming 'plans' is not a recent development, but there does not as yet seem to be a clear cut standard definition of what comprises a plan. Rich, Shrobe & Waters [1976] considered a plan as an abstract description of a program, which may be composed of subplans, or may itself be a fragment of another plan. Johnson [1986] defines plans as stereotypic action sequences in programs, used to map intended goals to code. Bonar & Weil [1986] refer to plans as the standard concepts and techniques for implementing common tasks chosen by the programmer in order to realise a higher level goal plan. Initially these plans would probably be only partially instantiated. Later the 'slots' of these skeletal plans would be fleshed out, the fillers for the slots perhaps being other lower-level plans which would also have to be coded and assembled. Rist [1986] discusses plans in terms of the 'focal' plan used to implement the main goal of the program. This focal plan then determines the number of other plans which have to be implemented. Again the principle is that a plan can be subdivided into lower-level plans, or can itself form part of a higher level plan.

The psychological importance of the role of plans in programming is also not clear cut, but remains an issue of debate. Spohrer Soloway &

Pope [1985] interpret their empirical studies as support for the development of a theory of novices' programming, in which the misuse of plans forms the underlying structure of programming problems. A mismatch between novices' intended plans and the plans they actually use to achieve their goals, omission of these plans, or a faulty merging of plans with the consequence of unwelcome or unexpected side effects, is seen to explain the majority of errors that occur in novices' programs. This view is not shared by Gilmore [1988], who points out that it cannot validly be claimed that plans represent the underlying structure of programming problems until empirical studies have shown that they play the same role in languages other than Pascal. This question of the role of plans in other languages is particularly relevant to Prolog and will be discussed in section [1.1.5]. There is no disagreement however about the importance of plans, in that they are seen as central to the process of turning an algorithm into a program [Johnson 1986].

Not only must the programmer have an awareness of the constructs of the programming language in order to form plans, but in order to assess the effect of the plans they use, they must also have some concept of how the computer executes them. It has been suggested by Du Boulay & O'Shea [1981] and by Du Boulay, O'Shea & Monk [1981], that problems are much more likely to arise if the novice has not formed a satisfactory model of program execution and empirical work by Mayer [1981] has given support to this theory. This does not imply that the student must know the complete workings of the computer down to machine level, but that the student should have some model of how the machine interprets the particular constructs that constitute the program. Without a clear idea of what operations the machine performs in response to the program that is input, or in what order these operations

are performed, a student is unlikely to be successful in program planning.

It can therefore be concluded from existing research that problems are likely to occur at the planning stage of programming since novices may well be unfamiliar with the constructs of the programming language and so be unable to formulate the necessary plans. In languages such as Pascal, work has shown that problems do often arise from a misapplication of, or faulty knowledge of programming 'plans'. Novices may also lack a satisfactory model of program execution, resulting in inappropriate planning or choice of plans.

1.1.3. Problems in program coding

Much of the knowledge about problems that students experience in coding their programs has been deduced either from in-depth protocols of novice students attempting to write their own programs or by error-analysis of programs written by novices. Both methods have features to recommend them and both have drawbacks. The former method is valuable, since it provides a rich source of students' attitudes, thoughts and difficulties as they actually write the program code, but it is expensive in terms of time and resources [Rajan 1986]. The latter method, i.e. analysis of completed code, allows the work of many more students to be assessed and broadens the numerical basis, in the light of errors found, from which conclusions may be drawn. Unfortunately this method also precludes first-hand evidence of the cause of many errors, so making the task of analysing data in a meaningful way a more complex one.

In spite of these drawbacks, research has produced some significant indicators of novices' problems at this stage of the programming process. The classification of the sorts of errors found in novices' code has rightly occupied much research time, the crux of the problem being the identification of the causes of those errors. Edward Youngs [1973], offers four broad areas of error: syntax, semantic, logical and clerical. This classification was based on a study covering five programming languages. It was extended by Du Boulay & O'Shea [1981] to include stylistic errors to cover, for example, instances of code which works but is very inefficient, or very hard to comprehend. According to Youngs, logical errors may be due to a misfit of the program to the problem, i.e. they are symptomatic of problems at the planning stage. Clerical errors may be relatively trivial, such as spelling mistakes, or programming syntax errors, the correction of which sometimes add little to an understanding of programming [du Boulay, O'Shea and Monk 1981]. Syntactic errors arise from incorrect use of the syntax of the language and would be indicated by error messages when the program is compiled or interpreted.

Semantic errors while not necessarily producing error-messages at run time, result in faulty or unexpected output and may be symptomatic of problems at a different level. They indicate that the student has a problem or problems that cannot be accounted for in terms of simple syntactical mistakes or clerical slips [Allwood 1986] and point to a misunderstanding of the semantics or the behaviour of a construct in the programming language being used. Research attempts have been and are being made to identify the potentially more serious problems indicated by this 'semantic' level of error. Such problems could relate to:

- fundamental misunderstandings of the language, [Brna, Bundy, Pain & Lynch, 1987]
- inaccurate models of program execution [Jones 1984]
- misapplication of plans [Spohrer, Soloway, & Pope, 1985]
- interference from pre-programming knowledge [Bonar & Soloway 1985]
- limitations of short term memory, as has been suggested by [Anderson & Jeffries 1985].

Clearly, we can conclude that problems experienced in writing program code, other than those related to difficulties of a relatively superficial syntactic sort, are intimately related to problems encountered at other stages of the programming process.

1.1.4. Debugging problems

As noted by Lukey [1981], debugging is closely connected to program comprehension since to do it successfully requires a certain level of understanding of the program being debugged. Program comprehension is an essential part of the programming process, used in correcting buggy programs and adapting programs. Much of what has been learned about the problems that novice programmers experience in comprehending and debugging programs, is based on comparative experiments between 'experts' and 'novices' [Adelson 1984], [Jeffries 1982], [Gugerty & Olson 1986]. In these experiments the groups are each given a particular program or section of a program to 'debug'. Unsurprisingly, results show that novices take longer and find it harder to debug programs than do experts, they are unable to make as much use of error messages as are experts [Davis 1983], experts possess much more information and that information is organized more efficiently [Mayer 1981]. As Kahney [1982] rightly points out, insights gained from comparative studies tend to give information about experts' knowledge while conclusions regarding novices tend to be of the default kind, that

they do not have whatever particular skill, knowledge, expertise it is shown that experts possess.

There are two factors which are often quoted as contributing to the higher success rate of experts in debugging programs. One is that experts possess a more sophisticated model of program execution, the other is that they have a more thorough understanding of the programming language [Jeffries 1982]. It is safe to assume therefore that basic problems mentioned in the previous section also have bearing on the difficulties novices experience in debugging. i.e. misconceptions concerning the language and lack of understanding of the program execution. Added to this, when they are debugging programs, novices have the problem of interpreting machine error-messages that for the inexperienced are sometimes unhelpful, sometimes obscure and possibly both [du Boulay & Matthew 1984]. Jeffries [1982] is the only exception to this in the literature. In this study she found that for a certain type of error message, e.g. "missing semi-colon on previous line", it was apparent that Pascal novices were very familiar with these error messages and were obviously quite experienced at tracking down the offending code. She attributes this to the relative frequency with which they inadvertently commit such syntax errors. This apart, error messages produced by the machine are often hard for novices to understand since understanding them can require more knowledge of the machine interpreter than novices possess.

1.1.5. Difficulties related to learning Prolog

We have looked at the literature related to learning to program. These studies have covered subjects beginning to learn not just Prolog, but other languages as well. We have seen that novice programmers have difficulties in finding algorithms to provide solutions to programming

problems and in producing coherent program plans from these algorithms. Difficulties stem from their misapplication of everyday problem-solving knowledge to the computing domain and possibly from their semantic confusion between natural language expressions and similar expressions in the formal languages of programming. They have semantic and syntactic problems in coding their program plans and difficulties in debugging the resulting programs should these produce error-messages at run-time, or if the output is incorrect. Studies of novice programmers have shown that these problems lie not only in beginners' comparative lack of knowledge of the programming language being learned and relatively small store of plans of how to achieve goals with the language constructs, but also in their lack of a model of program execution. The greater part of these difficulties seem to be common to most novices and not necessarily related to a specific language. To what extent do Prolog novices share these difficulties?

Before answering this question it seems sensible to clarify the term Prolog novices. It may be argued that the term 'novice Prolog programmers' should also include those who, although having considerable experience of other programming languages, have no experience of Prolog. Those used to an instruction oriented language may have difficulties in reconciling themselves to the declarative and propositional character of Prolog code. The control flow mechanisms responsible for the backtracking behaviour and 'matching' process in Prolog could give rise to problems for those who, even though relatively experienced programmers, are newcomers to Prolog. Work by van Someren [1984], [1987] and by White [1987], has shown that such students may well have particular problems, due to faulty transference of knowledge from more procedural languages. The transfer of

knowledge from another language is an interesting problem, but is not within the scope of this work and will not be dealt with further. While not specifically excluding this class of student from the issues discussed here, the term 'novice' will be used in this context primarily to refer to those students who are new to both programming and Prolog.

Prolog novices do seem to experience difficulties in much the same contexts as novices of other languages, but for Prolog novices the problems may well be more acute [Taylor 1987]. This is largely due to language-specific factors, which we will discuss in the following sections.

Firstly, the syntax of Prolog is based on predicate calculus statements formulated in terms very similar to those used in natural language reasoning, which increases the likelihood of natural language interference. Although one might think that a programming language which more closely approximates the natural language 'reasoning' used to solve problems, would help novices in the process of developing an algorithm, this is not necessarily the case. Research by Taylor [1987] has investigated the effect on novices of the surface similarity between natural language equivalents and the predicate logic used in Prolog. She concludes that rather than this similarity being a built-in advantage when students are developing an algorithmic solution to the programming problem, it can, on the contrary, lead to errors, since the semantics of natural language differ from those of formal logic. Studies by Johnson-Laird & Wason [1977] have also shown that although 'natural language' reasoning is in many ways similar to reasoning using predicate calculus, this does not hold in all cases. People presented with predicate calculus statements and asked to use these for a reasoning task,

are likely to make inferences which could be valid in a 'natural language' situation, but are not valid in formal logic. In natural language situations for example, use is made of implicit knowledge and conventional interpretations of quantifiers. In formal logic these factors do not come into play and a failure to appreciate this can be a source of confusion for Prolog novices.

Secondly and related to the similarity of Prolog syntax to natural language is the lack of easily recognisable programming constructs which can be incorporated in 'plans' to achieve program goals. As discussed in section [1.1.2], the application and integration of plans and sub-plans may well be a major source of errors by the novice programmer using Pascal [Spohrer, Soloway & Pope 1985] or any other programming language which has a syntax structure which lends itself to the formation of skeletal plans which can then be fleshed out in a slot and filler fashion. The situation is somewhat different in relation to Prolog. As a 'declarative' language rather than an instruction-oriented or functional language, the syntactic structure of Prolog does not lend itself readily to 'plans'. This does not mean that the novice Prolog programmer therefore has no problems, or no plans at the stage of programming planning. As Taylor & du Boulay [1986] make clear, at the planning stage students of Prolog have particular difficulty in transforming their solutions into programming plans. Unlike more obviously procedural languages, although the procedural aspect of Prolog programming is of great importance, its syntax does not offer any clear pointers to what is happening 'behind the scenes'. There are for example no structures such as 'while' loops or 'if ... then' statements which give perceptual clues to what operations are being performed as a result of the chosen syntax and which could help students to formulate

plans to achieve their programming goal. Strategies and techniques that are commonly used by Prolog programmers do exist but are syntactically and structurally more opaque [Ross 1987]. There are structures or "routines" that are standardly used to achieve certain results, such as construction and destruction of lists and current research work on "techniques" in Prolog [Brna et al 1988] is seeking to identify and clarify these. The syntactic difference between structures which produce very different results can however, often be slight and not obvious to a beginner. The predicate 'append' for instance, can be used to combine two given lists, when called with two instantiated arguments and a third uninstantiated argument, e.g. `append(+,+, -)`. It can also be used with two uninstantiated arguments and a third instantiated, e.g. `append(-, -, +)` to produce all possible 'splits' of a given list. To a novice the mode changes in these two uses of `append`, i.e. the use of the predicate with arguments instantiated or uninstantiated in a particular order, give very little indication of the contrasting purposes for which they would be employed.

Thirdly, it is possible to interpret the language in two ways, declaratively and procedurally. This is a feature of Prolog and merits a brief explanation. The declarative model of Prolog is one based on its development as a logic programming language. This allows the user to see a program as a collection of facts. These facts are stated in a subset of predicate calculus and from them other facts can be inferred to be true, or false, e.g. in the following program:

p if a & b & c.

and given the goal 'p', the declarative model would be

**The goal 'p' is true if and only if
there is an instance of 'p' in the program.**

**This instance of 'p' in the program is true if and only if,
'a' is true, 'b' is true and 'c' is true.**

A procedural model of the same program would be:

**We can satisfy the goal 'p'
if we can find a matching instance of 'p' in the program.
Having found an instance,
we must try and satisfy the subgoals,
firstly 'a', then 'b', then 'c' in order to satisfy goal 'p'.**

The former model is one of a logical statement, the latter of a list of goals and subgoals to be satisfied.

The declarative model represents the programming problem in terms of logical relationships which will allow the solution to fall out from the logical inferences that can be made. First examples are often based on family relationships of the following sort :

**sister (A, B) if
 female (A) &
 parents (C, D, A) &
 parents (C, D, B).**

There are advantages to this, since 'everyday' knowledge can be employed and it is possible for the novice to form a model of the

language relatively quickly (although work by Ormerod [1986] has cast some doubt on whether the typical family programs used in presenting the declarative model to beginners does in fact allow much transference of problem-solving skills to other problem situations).

The procedural model of the language is based on seeing Prolog as a goal-directed language. A program is seen as a process of goal satisfaction, rather than a collection of logical rules and facts from which inferences can be made. This model emphasises 'how' a solution is found rather than what logically constitutes a solution.

If the logic-based declarative view of the language is stressed, which is often the case when introducing beginners to Prolog, novices are often encouraged to view Prolog as natural language statements couched in predicate calculus terms. This serves to obscure the procedural aspect of Prolog [Ormerod 1986], [Rajan 1986]. Students are tempted to translate a natural language reasoning of the problem directly into Prolog code [Taylor & du Boulay, 1986]. Since Prolog has relatively few syntactic rules, it is possible for students to do so, unaware of the importance of such things as clause ordering and without having a clear idea of how the procedural 'machine' will act on this code. Not unsurprisingly, this leads to problems at a later stage when the resulting program produces unexpected results.

Finally, another important factor contributing to problems which are specific to Prolog novices, is the opaque nature of Prolog's procedural execution. There are very few syntactic markers to give novices clues to this procedural nature of the execution [Looi & Ross 1987]. The syntactic structures, such as the 'if then' or 'begin end' of more instruction-

oriented and procedural languages, which serve as clues to the machine's behaviour, are absent in Prolog. This lack of surface markers can make it much harder for students to form a clear model of program execution. The lack of a reliable model of execution for Prolog, which has an exceptionally powerful backtracking mechanism and matching process, is likely to hinder students' attempts to construct a program from their algorithm and to cause significant difficulties for them at the debugging stage.

From these research findings it is apparent that in addition to whatever other help can be given, it is important that Prolog novices are given help in forming reliable and useful models of Prolog's execution. This will be discussed more fully in the next section in which we look at help currently available to Prolog novices.

1.2. What help is available for Prolog novices?

Given that in studies of novice programmers there have been at least five classes of errors identified, syntactic, semantic, logical, clerical and stylistic [du Boulay & O'Shea 1981], it is apparent that there is ample scope for offering help to novices. It is equally apparent that the help must also address different aspects of the programming task. At one end of the scale as it were, help is needed for the 'clerical' type of errors, syntactical errors which are due to slips, oversights, lack of familiarity with a keyboard, or tiredness, the sort of errors referred to by [Hasemer 1983] as 'silly errors'. Time spent chasing a missing separator for instance, could usually be better spent. At the other end of the scale students need help in understanding the concepts of the programming language. In this context, 'concepts' of the language encompasses the role of variables, permitted syntactic structures and the type of

operations and programming techniques that can be performed in the language being learnt. Some of these concepts, recursion for example, are common to most programming languages while others, such as the role of variables, can be specific to the language. Closely related to this level of help and particularly relevant for Prolog novices is the help needed in developing a model of the machine interpreter, of program execution. In a brief overview of help which is provided for novice Prolog programmers and of research work related to extending and improving that help, we consider how the needs of the novice are met.

1.2.1. Prolog courses

Courses for Prolog novices are usually textbook-based, augmented by a combination of lectures and hands on experience. Usually there is also individual help provided in tutorials, which through constraints of time and expense are necessarily at set times and of a limited duration. For the rest of the learning experience, Prolog novices rely on whatever automated help is available. A more recent development has been the Integrated Prolog Course, developed by Eisenstadt & Brayshaw [1988] at the Open University, designed for distance learning, which incorporates textbook, audio and video-taped sessions. These teaching materials are intended for integrated use with a Prolog environment package [Eisenstadt & Brayshaw 1987] which will be discussed later in this chapter. Central to this course, is the goal of helping the student to develop a useful and reliable model of Prolog execution, which is reinforced in each type of teaching material being used. At this point it is perhaps useful to expand a little on what is implied by a useful and reliable model of execution.

The need for the programming student to form a representation of the machine's behaviour on which they can draw when planning or debugging a program has been supported by, among others, du Boulay, O'Shea & Monk [1981]. In their work they describe this modelling of program execution as the development of a conceptual model of the 'virtual' machine. This is similar to the "conceptual" model described by Young [1981], but in the case of learning to program it is not in fact the behaviour of the machine itself which the user is normally attempting to understand, but the behaviour of the machine at the level at which it is interpreting the programming language being used. If Lisp is being run, or Pascal, or Prolog, this behaviour will differ accordingly, hence the term 'virtual' machine.

On the majority of Prolog courses, the model of program execution, that is usually provided for students to help them understand the 'virtual' machine, is one of, or a combination of, the following three: a Byrd box model, an arrow diagram model or a tree model. We will look briefly at these and the way in which each attempts to provide a model of the actions of the Prolog interpreter during execution of a program.

The Byrd Box Model

This is the model upon which the conventional Prolog trace package is based. Developed by Byrd [1980], the program execution of Prolog is seen as a series of boxes denoting the goals to be satisfied. Control flow is represented by a path through entry and exit ports in these boxes. On a goal being called at predicate level, the goal 'box' is entered from the left. If the goal conditions are satisfied, the box is exited to the right - the goal has succeeded. If the goal fails, it is retried, the box being re-entered from the right. If this 'redo' fails, then the box is exited by a 'fail' port on

the left - the goal has failed. The simple program and query below, followed by the equivalent Byrd box model, will illustrate this.

program: sister (ann, bill)

query: ?-sister (ann, bill)

To satisfy this goal, flow of control enters the 'call' port, succeeds and leaves via the 'exit' port. If it had failed, because the fact had not been found in the data-base, the redo port would have been used. If the redo had failed, then flow of control would have left the goal box via the 'fail' port. These four ports, 'call', 'exit', 'redo' and 'fail' form the basis of the control flow description.

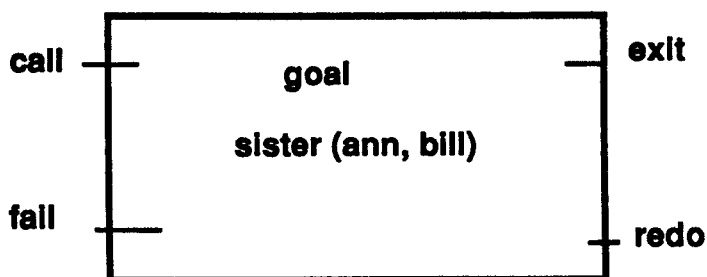


Fig.1.2. Byrd Box Model of simple Prolog goal.

In the case of goals which have subgoals, these are represented as boxes nested within boxes. If for instance the goal above had been a subgoal of a higher level goal, such as in the following program:

aunt(X,Z) if

sister(X,Y) &

parent(Y, Z).

where the parent goal calls on two subgoals, then given the goal query:

?- aunt (ann, jill)

using the Byrd box model this would be represented as shown below:

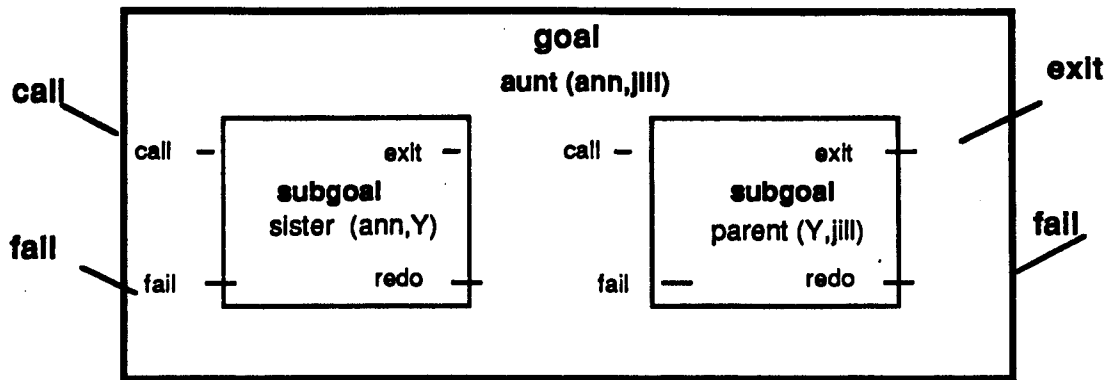


Fig.1.3. Model of embedded goals.

The Arrow Model

Representations of this model may vary to some degree, but the modelling principle is basically the same, in that it consists of using arrows to show the flow of control from goal to goal through the program [Clocksin and Mellish 1981] , or 'swinging' arrows [Pain and Bundy 1987] to indicate the state of the goal stack in relation to the database. An illustration of its use is shown below in the program:

aunt(X,Z) if

sister(X,Y) &

parent(Y, Z).

in which the query:

?- aunt (ann, jill).

is matched with a rule, causing the goal stack to expand to include the two subgoals, sister (ann,Y) and parent (Y, jill), these two subgoals then being matched against facts in the data-base.

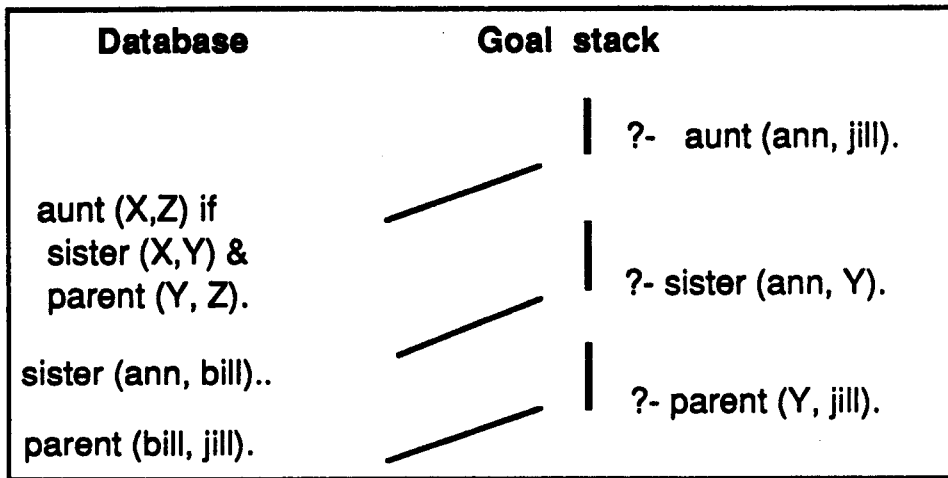


Fig.1.4. An Arrow Model of Prolog control

This model is a good one when used with simple goals, showing clearly the relationship between the goal search and the database, though when used for problems of any complexity it could become difficult to illustrate clearly the global flow of control.

The And/Or Tree Model

The parent goal is the 'root' of the tree, which branches out according to the number of subgoals that must be satisfied in order to satisfy the parent goal. These branches are traversed depth first, left to right.

Conjunctive goals, i.e. 'and' branches, are usually represented by connecting arcs between them, indicating that the goals represented by all the connected branches must be satisfied, while disjunctive goals, 'or' branches are recognised by the lack of these connecting arcs, indicating that it is sufficient for one of these branches to be followed successfully.

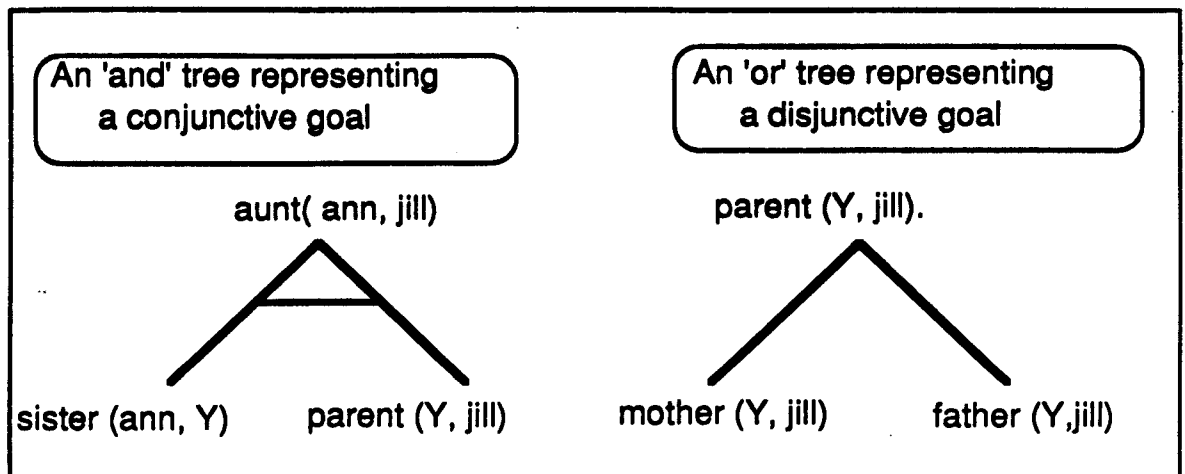


Fig.1.5. A Tree Model

In the first tree, both 'branches' must be successfully instantiated in order to satisfy the parent goal 'aunt (ann, jill)' whereas in the second, the parent goal will succeed if either one or the other of the branches result in success.

This representation is the one favoured by the Edinburgh Prolog group as a model which offers the best opportunity to novices to develop a simple but informative model of the Prolog interpreter. The work of this group on Prolog 'stories' [Bundy 1984], [Bundy, Pain, Brna & Lynch 1985] [Pain & Bundy 1987] focuses on the need to offer students a model of Prolog execution, i.e. the 'story' which students should be told to help them understand the procedural semantics of Prolog. In their work they stipulate that ideally it should be one which:

- covers all the important aspects of Prolog behaviour, so that it can be safely used to predict the behaviour of Prolog programs
- is simple to understand and use, even by people with no previous computing experience
- would illuminate the tricky aspects of Prolog behaviour such as pattern directed invocation, backtracking
- would be used universally by Prolog teachers, primers, trace messages, error messages etc.

While progress has been made towards this goal by such courses as the Integrated Prolog Course mentioned above [Eisenstadt & Brayshaw 1988], the reality of Prolog courses does not always match this ideal. Apart from any other considerations, 'live' help for novices in the form of experienced Prolog teachers is an expensive resource, not freely available in any sense of the word. The average novice faces hours of 'on-line' experience without necessarily receiving any 'live' help in that time. Any available automated help is likely to play an important role in the learning process. At present this is principally help given by the programming environment to which students have access. In the following section we look at this help. The term automated help includes computerised tutoring help. This latter type of help will, however, be discussed in a subsequent section. This decision to differentiate between environmental help and computerised tutoring is based on the judgement that the overall difference in their respective aims and objectives merits such a distinction.

1.2 2. Prolog environments

The term environment is used here to include the editing tools offered to the student, the version of Prolog being used and its incorporated automatic debugging tools. The basic role of tools in automated help for novices is to make it easier for students to design better programs and to help them find errors when a program fails to produce correct or expected results. A programming environment would normally include editing tools and static or dynamic debugging tools. A 'static' tool usually refers to a process of code-analysis which takes place at compile time and a 'dynamic' tool to an analysis applied when the program is being executed. The distinction is not clear cut, since some debugging systems such as [Eisenstadt & Brayshaw 1987], store the results

of the 'dynamic' analysis and this is consequently used in a 'static' analysis. The dynamic tools are usually incorporated in the trace package being used with the Prolog implementation.

Research by Brna, Bundy, Pain & Lynch [1987], undertaken to construct a coherent framework on which to base the design of programming tools, came to the conclusion that their investigation of available debugging strategies had been fruitful in terms of revealing the shortcomings of existing Prolog environments. There are two factors which could contribute to the validity of such a conclusion, which we will briefly consider before going on to look at the tools supplied in Prolog environments. One factor, though perhaps of secondary importance, is the relative recency of the Prolog programming language. Developed by Alain Colmerauer and his group at Marseilles in 1974, the first textbook for beginners [Clocksin & Mellish 1981] is not yet ten years old. Prolog has simply not gone through as long and as wide a developmental process as many other longer established programming languages. Accruing research in the area of novice Prolog programmers, of which the above mentioned paper is part, is beginning to form the basis of developments taking place, but this process takes time.

The other factor, probably of primary importance, lies in the structure of the language itself, in its lack of syntactic markers, as was discussed earlier. Syntactic 'markers' are useful in developing automatic debugging tools, and the relative lack of them in Prolog is a disadvantage in such developments. Work to develop tools which would recognise 'plans', or 'techniques' [Brna et al 1988] is made difficult, since due to the backtracking and unification features of the Prolog interpreter, the same, or very similar syntactic structures are used

to achieve different results, as was pointed out earlier in the contrasting uses of the 'append' predicate. Work by Payne, Sime & Green [1984] on perceptual cueing has indicated that relatively minor changes made to a program by a text editor, (in their experiment one such change was to put operation codes into upper case), could significantly decrease error frequency. It is not clear that in Prolog such developments would necessarily be of the same value to novices, since there are relatively few structural syntax cues which could be highlighted or emphasised in some way to help students pick up errors when writing or checking their code. These are factors which must be borne in mind in assessing the quality and development of tools designed for use in programming environments for Prolog.

Editing Tools

There are text editor features which can be of great help, such as parentheses balancers plus good text manipulation facilities in general. A powerful example of this is the Prolog mode of the emacs editor which offers features such as automatic indenting and easy loading of marked sections of code. These and other facilities such as multi-windowing to allow easier reference to the program text during execution, should be considered essential in helping to build a suitable environment for novice programmers. Unfortunately this level of editing tools is, as yet, rarely an integral part of the Prolog environment (MacProlog being a notable exception). Provision of such editing tools usually depends to a large extent on the resources available locally. It must be concluded that for many Prolog novices, editing tools leave much to be desired.

Debugging tools

Static tools have been developed to do code-analyses which at compile time check modes i.e. whether the expected parameters of a predicate are instantiated or not [Mellish 1981]; check types, i.e. whether parameters are of the expected or necessary data type, for example, list, numerical or atom; check dataflow, syntax-errors and even look for possible typographic errors [Looi 1986]. The average Prolog environment however provides only a syntactic check to find errors such as missing separators or unmatched parentheses, leaving other errors, such as missing predicates or wrong arity to be detected at run time by the dynamic tools incorporated in the trace package. An exception to this is Quintus Prolog, which in conjunction with an emacs editor in Prolog mode can offer not only powerful editing, but also style-checking. With this system for instance, a single occurrence of a variable would be signalled, e.g. a variable occurring in a clause head and not re-appearing in the clause body. As an optional facility at run-time, a failed goal due to a predicate not found would result in an automatic drop into debugging mode. Dynamic tools are usually included in a 'trace package' based on a control flow trace of program execution and to varying degrees are designed to be interactive. The extent of the interaction ranges from the 'oracle' type interaction developed by Shapiro [1982], in which the system diagnoses errors acting upon information requested from the user, to systems which simply ask the user if she wishes to continue the trace. The degree to which the trace package can help novice programmers is equally varied, since some are potentially far easier to interpret than others. An overview of debugging tools for Prolog by Brna, Brayshaw, Bundy, Elsom-Cook, Fung & Dodd [1988] illustrates this point amply. Below we discuss a selection of trace packages.

Trace packages

The first case looked at, using the Byrd box model, was chosen because it represents the automated help that almost every beginner is likely to encounter. Others, even though only prototypes or not yet widely available, have been selected because they represent stages and improvements in the development of automated tracing tools which are likely to be more helpful for novices. As Lieberman [1985] has pointed out:

Watching a program work step-by step, where each step is reflected in visible changes to the display screen, greatly facilitates understanding of the internal workings of a program.

This can equally be applied to Prolog trace packages, but the operative word must be 'visible', since the visibility, or transparency of some execution traces is far superior to others.

Byrd box based trace packages

The trace packages most widely known and available for Prolog at present, are built around the Byrd box model of execution illustrated graphically in section [1.2.1].

Their trace outputs show the control flow of the program in terms of the 'ports', usually the four illustrated, the call, exit, redo and fail ports. A fragment of a typical example of trace-output from this type of trace package is shown below (taken from the trace of a sorting program 'qsort').

```

| ?- qsort([2,1,3],P).
(1) 0 Call: qsort([2,1,3],_501) ?
(2) 1 Call: split([1,3],2,_619,_620) ?
(3) 2 Call (built_in) : 1<2 ?
(3) 2 Exit (built_in) : 1<2 ?
(4) 2 Call : split([3],2,_681,_620) ?
(5) 3 Call (built_in) : 3<2 ?
(5) 3 Fail (built_in) : 3<2 ?
(6) 3 Call (built_in) : 3>=2 ?
(6) 3 Exit (built_in) : 3>=2 ?
(7) 3 Call : split([],2,_681,_753) ?
(7) 3 Exit : split([],2,[],) ?
(4) 2 Exit : split([3],2,[],(3)) ?
(2) 1 Exit : split([1,3],2,[],(3)) ?
(8) 1 Call : qsort([1],_629) ?
(9) 2 Call : split([],1,_879,_880) ?
(9) 2 Exit : split([],1,[],) ?
(10) 2 Call : qsort([],_889) ?
(10) 2 Exit : qsort([],) ?
(11) 2 Call : qsort([],_898) ?
(11) 2 Exit : qsort([],) ?
(12) 2 Call : append([],[1],_629) ?
(12) 2 Exit : append([],[1],[1]) ?
(8) 1 Exit : qsort([1],[1]) ?
(13) 1 Call : qsort([3],_638) ?
(14) 2 Call : split([],3,_1092,_1093) ?
(14) 2 Exit : split([],3,[],) ?
(15) 2 Call : qsort([],_1102) ?
(15) 2 Exit : qsort([],) ?
(16) 2 Call : qsort([],_1111) ?
(16) 2 Exit : qsort([],) ?
(17) 2 Call : append([],[3],_638) ?
(17) 2 Exit : append([],[3],[3]) ?
(13) 1 Exit : qsort([3],[3]) ?
(18) 1 Call : append([1],[2,3],_501) ?
(19) 2 Call : append([],[2,3],_1298) ?
(19) 2 Exit : append([],[2,3],[2,3]) ?
(18) 1 Exit : append([1],[2,3],[1,2,3]) ?
(1) 0 Exit : qsort([2,1,3],[1,2,3]) ?

```

Fig.1.6. Sample standard Byrd box trace

Each line of the trace displays (left to right):

- i a number, uniquely associated with a particular goal call and used each time a port in that goal call is used
- ii the depth of that call within the execution of the program
- iii the name of the port currently being used
- iv the call and its parameters.

For novices there are several disadvantages to this trace model, these are discussed in detail by Dewar & Cleary [1986]. The most obvious are the amount of detail displayed and the strictly linear display of information.

While appreciating the amount of information that it contains and the help that this can be in debugging a faulty program, for a novice it can be simply confusing to see line after line of similar trace. Although the information is there, it is presented in such a way that there are no immediate visual cues which would enable the novice user to relate the trace to the structure of the program. Embedded calls are uniquely numbered, but to follow the path of those numbers from their original calling to their satisfaction or failure, meanwhile bearing in mind their significance in the overall structure of the program, requires a degree of experience in debugging that novices could not be expected to have acquired.

There are usually options of restricting output to certain of those ports, of 'skipping' through the output or inspecting it a step at a time, this varies from implementation to implementation. Taking advantage of these options allows the user to 'spy' or trace only one or some predicates, which reduces the overwhelming amount of trace information output on screen. This does presuppose that the user has a good idea of which clause or part of the program is either causing an error or producing unexpected output. In addition, to interpret the significance of the path of just one or two predicates taken out of the context of a whole program trace requires the user to be in possession of a good working model of the program's overall execution path, again, a requirement that often cannot be met by novices. Although an invaluable aid for programmers once they have enough experience to make use of it, the Byrd box model based trace as it is most often encountered is a tool that is not significantly helpful for novices.

Improved trace packages

While trace packages such as the model described above are widely available and used, research and development has gone ahead on producing trace packages which will either minimise or eliminate the problems mentioned, making them more of more practical help for beginners. As pointed out earlier, it may well be simply a matter of time before other trace packages for Prolog, based more directly on what research has shown novices and programmers need, will be developed and become widely available. The direction of certain of these developments are outlined below.

Staying with the Byrd box model of program execution, [Eisenstadt 1985] has produced an improved Prolog Trace Package (PTP) [Eisenstadt 1984], a means of tracing and debugging Prolog programs by 'retrospective zooming'. This system stores an exhaustive trace of the program execution and subsequently subjects this information to an analysis with the goal of detecting 'suspicious clusters', i.e. sections of the trace output which could be indicative of an error. This section only can then be 'zoomed' in on by the user. Several extra symbols are introduced into the trace, including the provision of more informative categories of 'fail' and 'succeed'. This system, although in principle much more sympathetic to the needs of the user, suffers from some of the faults of the standard trace. The output is still linear and while increasing the amount of information given, it involves, in the case of novice users, the trade-off of interpreting the added trace symbols. A small section of trace from the improved PTP is given below.

```

PTP: qsort([2,1],R).
1 : ? qsort([2,1],_44)
18 : - qsort([2,1],_44) (2)

PTP: retrotrace.
1 : ? qsort([2,1],_44)
2: > qsort([2,1],_44) (2)
3: ? split([1],2,_44,_45)
4: >split([1],2,[1]_44_45) (1)
5: @ 1<2
6: ++1 <2
   &
7: ? split([],2,_44,_45)
8: - ~split([],2,_44,_45)

```

Fig.1.7. Sample trace of Prolog Trace Package

Other developments directly based on the Byrd box model, which seek to improve and extend the quality and visual impact of its output include work by Plummer [1987] on the CODA system (Clause Oriented Debugging Aid) and work by Dichev & du Boulay [1988] on a data tracing system for Prolog. Both are concerned with enriching the model of Prolog which is presented by the trace output. Plummer's system is designed to extend the control of the user over information shown about the unification process, in that the user can interactively determine the effect of instantiations in chosen clauses as they are made. The main thrust of Dichev & du Boulay's work is twofold. One aim is to present the trace output in a way which will show information in a format that is more meaningful to novices, relating it to the matched clauses and original variable names. The other is to make it useful to beginners in terms of building a model of the unification process and of understanding how through unification the data manipulation which takes place in executing a Prolog program is achieved. One of the conclusions which arose from their discussion of design principles and the initial implementation of such a system was that a more

diagrammatic format may have a significant contribution to make to this approach. This could alleviate one of the main problems associated with developments based on the Byrd box trace, i.e. the volume of textual information which is produced and the difficulties this presents for novices trying to interpret that information.

Rajan [1986] has developed a prototype debugging tool, APT (Animated Prolog Tracer), constructed according to design principles he has formulated for animated tracing of program execution. It is based on the arrow model described above in section [1.2.1] rather than the Byrd box model. Aimed at novices, its goal is to give the learner a dynamic view of program execution, showing the matching process and the instantiation process as the program proceeds. Database information and the current state of goal execution are displayed in two separate windows. Database rules or facts in the upper window are highlighted as they become matched with the current goal, which is displayed in the lower window, where the resulting instantiation of variables is also displayed in inverse highlighting. A status line reports on the current state of execution. At present not developed beyond a prototype, it is a very interesting example of a tracing tool which actively reinforces a conceptual model of program execution. Results from a first study of differences between novices using APT and a control group not using APT have been encouraging [Rajan 1986]. They show that there is a statistically significant improvement in the ability of novices to solve queries to Prolog programs after seeing an animated demonstration of program execution. This gives reason to believe that the novices using APT also have an improved conceptual model of the action of the Prolog interpreter.

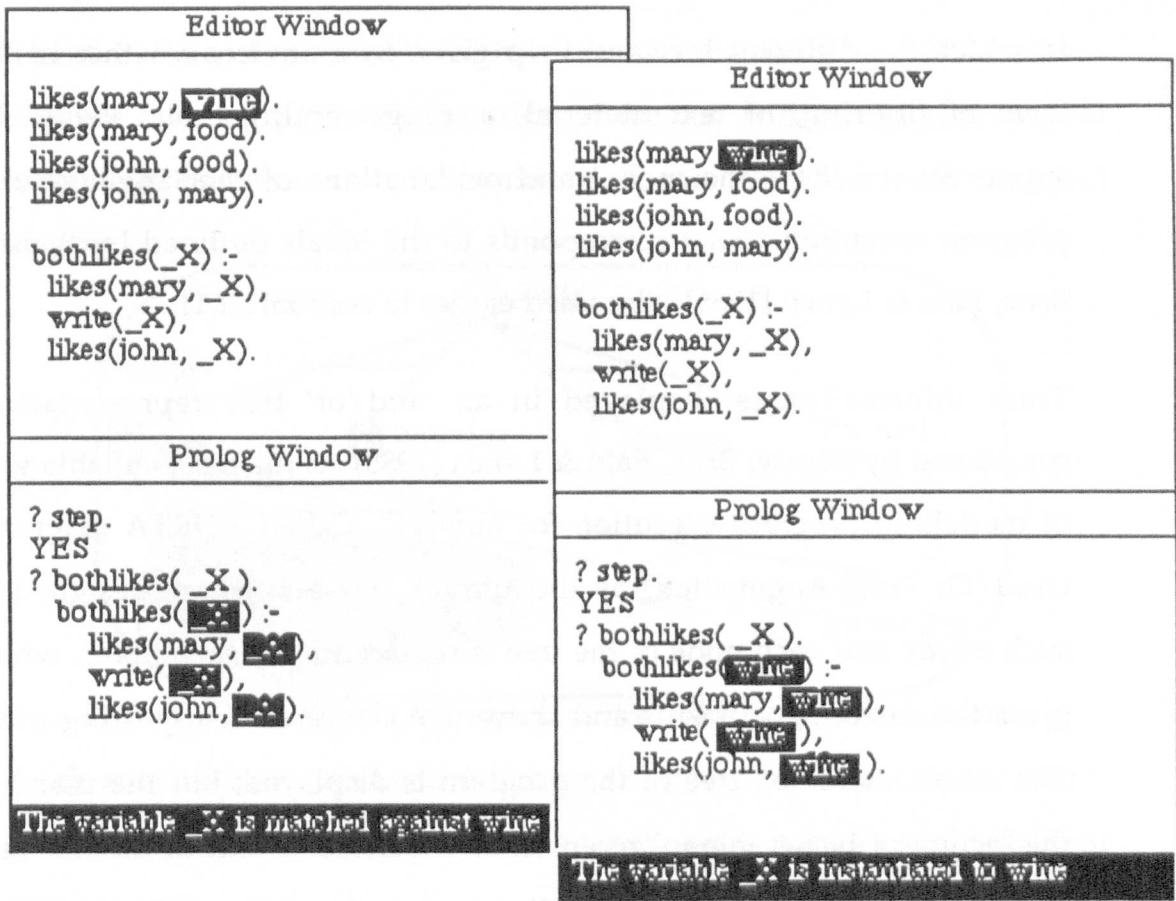


Fig.1.8. Snapshot of APT tracing tool

Continuing work towards a trace package which would be yet more comprehensible to the user and more supportive of a conceptual model of the language, led to the development by Eisenstadt and Brayshaw [1987] of the 'Transparent Prolog Machine' (TPM), a trace package which incorporates a graphical debugger. With increased availability of graphics and workstations which support them, there will almost certainly be a growing interest in graphical tracers, but as yet there has otherwise been relatively little work undertaken for graphical debugging tools for Prolog. Work on graphical debugging by Dewar & Cleary [1986] in this direction has not yet reached the stage at which their system 'Dewlap' could be considered a production tool. The TPM system is designed as a tool to be used by both novices and experienced Prolog programmers. It is, however, a major contribution to an environment

in which the different levels of help given to a novice, whether in the form of tutoring, of text material or programming tools, would be supportive of the same model and explanations of the language and program execution. This corresponds to the ideals outlined by Bundy, Brna, Pain & Lynch [1985], discussed earlier in section[1.2.1].

Trace information is displayed in an 'and/or' tree representation, considered by Bundy, Brna, Pain & Lynch [1985], as the most suitable way of modelling program execution for novices. Called AORTA diagrams (And/Or Tree, Augmented) by the authors, these trees are designed in such a way that each node of the tree is replaced by a 'status box', which gives the current goal status and shows the clause currently being tried. The whole execution tree of the program is displayed, but the user has the facility of being able to 'zoom' in on a particular part of the tree and open up a node and investigate all the execution information available at that point. This tracing tool was designed to be used as part of a text-book and video-supported distance learning package, and its model of program execution is clean, clear and consistent with all the teaching material contained in the course. A full description of TPM is given in [Eisenstadt & Brayshaw 1987]. It seems clear that this represents a milestone in the development of tracing tools for Prolog. In addition to the course participants for whom it was originally developed, its use is currently planned in several major universities and evaluations should start in the near future.

The three pictures below will give the flavour of a few of TPM's features. The first shows the overall tree model of control flow for a simple program determining the conditions for holding a party, i.e. either because one is happy and has a birthday, or to cheer up a sad

friend. Notice the 'cloud' which represents the frozen variable values of goals satisfied prior to invocation of the cut, and the scissors symbol indicating the removal of remaining clause branches from the goal 'happy'.

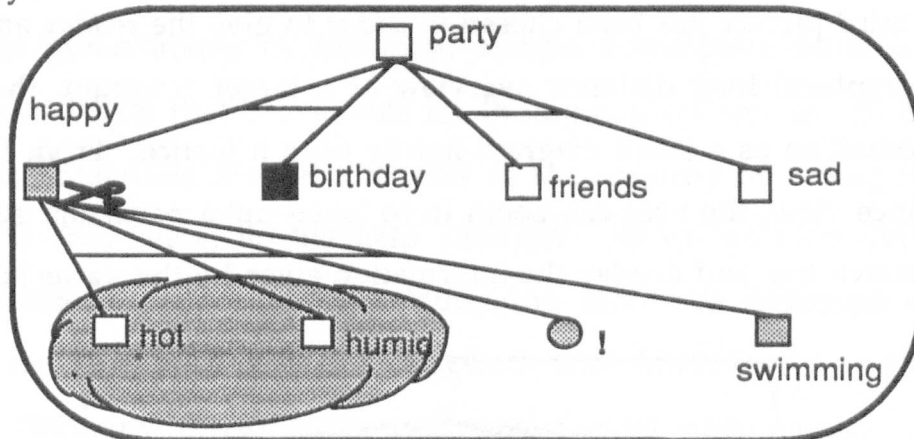


Fig.1.9. TPM trace of a program at the point of goal satisfaction

Black status boxes represent failed goals, white boxes show successful ones, and grey boxes represent goals which were initially successful but consequently failed.

In this second picture, below, we show a close-up view of the status box, which records the failure or success of the goal, how many clause branches it has and which clause branch is currently being tried. The symbols are all designed to be interpreted as intuitively as possible.

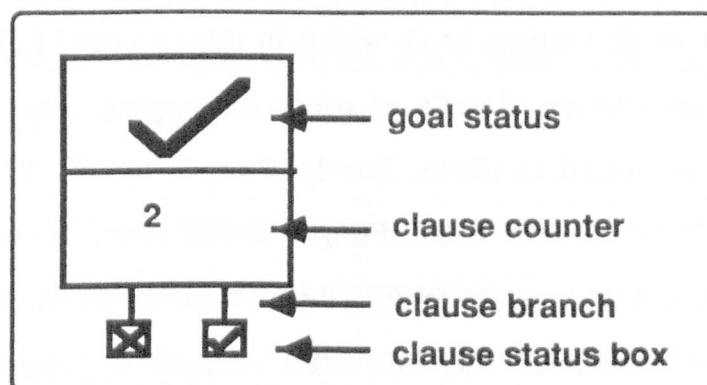


Fig.1.10. Detail of TPM status box

Optionally, the goal and its current instantiations can be displayed alongside the box. The variables shown would be those defined by the user.

The third picture has been chosen in order to give the reader an idea of the graphical long distance overview of a larger program, though its reproduction as a small diagram hardly does it justice. From this long distance view, the user can zoom in to 'open-up' a particular section of the search tree and display the information given by the status boxes.

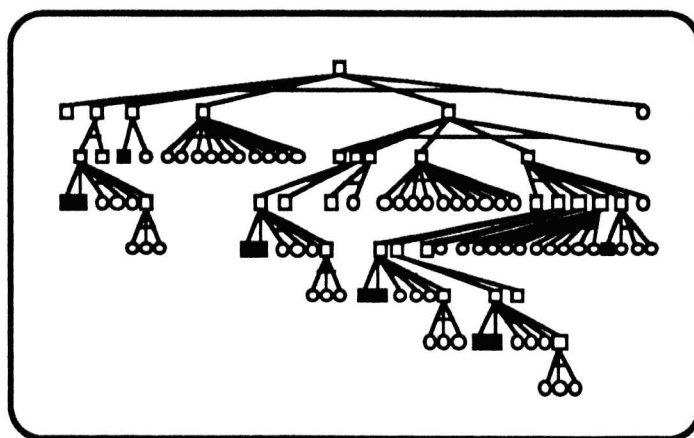


Fig.1.11. Long distance view of TPM trace

1.2.3. Summary

This outline of debugging tools which in theory could help novices is by no means exhaustive, details of other debugging tools developed for Prolog can be found in [Brna, Bundy, Pain & Lynch, 1987] and [Brna, Brayshaw, Bundy, Elsom-Cook, Fung & Dodd 1988]. Some of those tools have stood the test of time but have been of limited direct use to beginners. Sometimes this has been because a prerequisite of their effective use is a level of programming knowledge beyond that of the average beginner, as is needed for interactive debuggers of the 'oracle' type [Shapiro 1982]. Sometimes it is because the computer facilities

available at the time limited their usefulness, such as the semi-graphical trace package developed by Mellish [1984]. Others have been refinements of the basic Byrd box trace, but have not overcome the basic drawbacks, discussed above, which this trace holds for novices. The outline given above is, however, almost a complete outline of the automated help that is available for Prolog novices and not all of those systems described are yet available to the majority of Prolog novices. Even if they were, is this help enough? As we pointed out at the beginning of section [1.2.2], debugging tools are designed to help programmers construct better programs with fewer errors and to help them find any errors they may nevertheless have made. Novice programmers need more than this. They need guidance in how to interpret and use the information provided by the tools, and in identifying what information is relevant and what is not. When a novice makes an error, an explanation of why it is an error and some insight into what misunderstanding led to the error are extremely important elements of the learning process. Debugging tools are essential, since they allow errors to be detected, but in terms of automated help for beginners, their scope is necessarily limited, since they are designed to reveal errors rather than explain or investigate them. For experienced programmers this is usually enough, for novices this is rarely so.

It is at the point of discovering an error that beginners would most benefit from tutorial guidance or explanation. At present this help is only available for Prolog novices from their text-books or tutors. The limitations of this help means that novices can spend frustratingly long periods of time at a terminal without necessarily finding the underlying cause of an error, or how to correct it. It is a difficult task for a beginner

to relate the general principles of a programming language, set out in a text-book, to the particular problem or error which she or he is faced with on screen, whilst tutorial help, which is most needed at such a point, is a relatively scarce commodity, often only available on the scale of an hour or so a week. From a practical point of view too, since learning to program entails using a computer, it is not sensible to have students sitting at terminals in need of tutoring, and yet to leave untapped such a source of teaching potential as the computer itself.

There is then a need for automated help, in addition to the debugging tools currently available, which can provide some of the much needed expertise of a tutor. The growing body of research on the role of computers in learning and in education [Scanlon & O'Shea (ed) 1987], [Jones, Scanlon & O'Shea (ed) 1987] indicates that the development of computer tutoring is probably one of the most challenging areas of current educational research in computing. The idea of a computer tutor taking the place of a human teacher is one that has not proved to be practical, as shown by the American experience [O'Shea and Self, 1983]. It is doubtful if this would even be desirable. Nevertheless it is of great benefit when even a measure of an experienced human tutor's expertise can be formalised and made available on line for learners.

When interest in Prolog as a logic-based programming language of the future gained ground, it was hoped, if not expected, that its superficial likeness to natural language would eliminate some of the problems faced by novice programmers. It has only been in recent years that consideration has been paid to the fact that while Prolog has simple structures and syntax, its powerful procedural execution and its very similarity to natural language poses problems for novices [Taylor 1987].

The need for tutoring in Prolog is every bit as essential as in other languages, but only recently has attention turned to the task of providing some of that tutoring on-line [Coombs & Stell, 1985], [Looi & Ross 1987], while for other languages such as Pascal and Lisp, work on computer tutoring systems has already produced some interesting and impressive results. Compared to these longer established programming languages, Prolog is a relatively recent development. This too may be a contributory factor in the relative lack of development in tutoring systems for Prolog. In the following section we will look at some of the existing computer tutoring systems for these other programming languages and at the implications of this for designing tutoring help for Prolog novices.

1.3. Tutoring systems

In the development of tutoring systems some domains have proved to be more amenable to formalisation than others, since they provide very precisely formed knowledge boundaries and problems and solutions have much more well defined limits. This holds true for the field of electronics [Brown & Burton 1975], physics [Scanlon & Hawkrige 1984] or mathematics [Young & O'Shea 1981], [O'Shea 1982]. In such domains, even where the problems of student-tutor interactions and student-modelling have not yet been fully addressed, benefits from factors such as group discussion of simulation models have made their development a worthwhile goal [Laurillard 1978].

Other domains have more imprecise boundaries, and solutions to their problems are much more open-ended. In many respects learning to program is such a domain. Rather like using natural language, there are more ways than one of writing a program to achieve a particular result.

A given problem may have any number of "correct" solutions [Elsom-Cook 1986]. On the other hand a programming language is a formal language with constraints which may not be violated, at risk of the machine producing an erroneous result or no result at all. This open-endedness combined with a need for precision poses problems for tutoring strategies and implementation problems, which existing systems for tutoring programming languages have approached in various ways.

Several comprehensive and accurate surveys of important developments in the field of computer tutoring have now been written [Sleeman & Brown (ed) 1982], [du Boulay & Sothcott 1987], [Wenger 1987] and we will not attempt to replicate these here, but rather discuss a selection of tutoring systems in order to give an outline of the work and illustrate approaches which have been taken. It is worth bearing in mind a point made by du Boulay & Sothcott [1986], that at present no one tutoring system definitively addresses all the major issues in computer tutoring, such as teaching strategies, student modelling or subject representation, but rather each tackles some aspects of certain issues with varying degrees of success. Neither do they all address the same stages of learning, some being designed for complete beginners, others for students who are able to write syntactically correct programs. For convenience the tutoring systems discussed in the following sections have been grouped according to language, although of course many of the points discussed are applicable across languages.

1.3.1. Lisp tutors

The programming language Lisp has a range of examples of development of on-line tutoring aids to choose from, among them,

Talus [Murray 1985], Struedi [Wender, Weber & Waloszek 1988], Greaterp [Anderson & Reiser 1985], Impart [Elsom-Cook 1984] and Trill [Cerri, Fabbrizzi & Marsili 1983], [Cerri, Elsom-Cook & Leoncini 1988]. The latter three systems have been chosen for discussion here, since all three address the problem of tutoring complete novices, while each takes a different approach to this task. Greaterp differs somewhat, in that its structure presumes that the learner will have access to relevant teaching-text written by the same authors.

Greaterp

The original system takes a beginner through the first stages of programming, teaching the student how to write correct programs. Programs are written step by step, helped with templates of Lisp functions, which then have to be filled out correctly. A set of production rules for a 'correct' solution of the programming problem and a set of 'buggy' rules, based on past experience of common mistakes, are used as a framework for teaching and monitoring the beginners progress. Each section of code input by the student is immediately evaluated. If student input matches a 'correct' production rule, then the lesson goes ahead. If it does not and a match for the incorrect code is found among the 'buggy' rules, then advice is given. This advice is based on templates related to the suspected error. Following this the student is given another opportunity to try and input correct code. Menu-driven dialogue controls the interaction between student and system. If the system cannot interpret the student input, then she is reminded of the current goal and asked if she would like advice. This is constrained to a choice of receiving more explanations, seeing examples or starting again. At no point is the learner allowed to continue if her program is incorrect.

The teaching strategy is completely "top-down", the student being given a task and monitored closely to ensure that it conforms to the expert's model. Several disadvantages of this method are apparent. One is that it is difficult for the system to have an overall appreciation of the student's problems, since the learner is corrected at the first occurrence of an error or deviation from the system solution. Another is that it makes no allowances for different learning styles. Work by Pask [1976] and by Coombs, Gibson & Alty [1982] has suggested that there are important differences in learning styles. The significance in this context lies in that while the operational approach of learning to program step by step adopted by Greaterp may suit some students, it may be unhelpful for others. As opposed to those who may appreciate the security of being led step by step, there are those for whom this serial approach could be unduly frustrating and who would benefit from more freedom to attempt the programming task in its entirety. Such students may well benefit from an opportunity to experiment and learn from their mistakes.

The top down step by step approach also raises the question of the comparative educational merit of preventing the novice building a syntactically illegal program as against explaining what is wrong if she does so [du Boulay & Matthew 1984]. The authors justify their teaching strategy on the grounds that it is better to use correct and only correct code right from the start, and that the immediate interrupt mode is in keeping with the observation that "humans learn better with immediate feedback" [Anderson, Farrell & Reiser 1984]. This strictly monitored top-down approach also has significant benefits at implementation level. The student is never allowed to offer more than

one faulty input at a time, which greatly reduces the search space for errors.

Also a problem is the fact that emphasis throughout is laid upon the student achieving a syntactically correct program. The system is designed in such a way that there is no possibility of modelling the student's learning over the complete programming task, since each line of code must be correct before the student may take the next step. The tutor cannot address the underlying semantics of the language or problems associated with a misunderstanding of these. Another major difficulty is that although 'buggy' code is recognised and matched by the catalogue of 'faulty' production rules, the system itself has no representation of its 'knowledge' about program code. Information about the program being used is prestored. Code input by the learner which is syntactically faulty and code which while syntactically correct does not match the prestored solution are considered equally erroneous.

Nevertheless it is a system that is operational and the content of which is based on many years of teaching novices to program in Lisp. Pragmatically viewed, in spite of the limited teaching strategy and relatively narrow concept of programming that is offered to the learner, it must rate as an important source of help to novices.

More recent work on the system is designed to incorporate changes which allow students to choose whether or not to continue in spite of being notified by the system that their input is incorrect. The implications of this for the student modelling process are not yet clear.

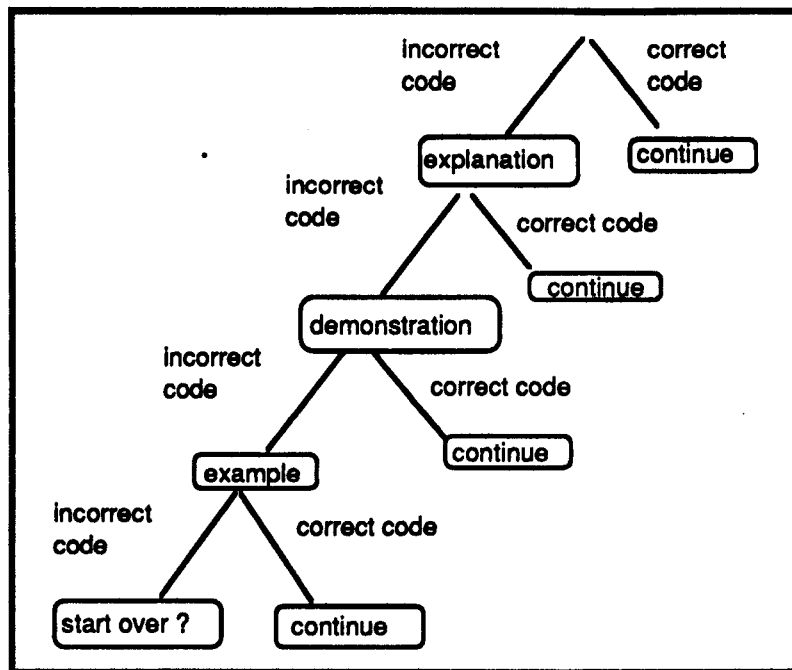


Fig.1.12. Approximation of teaching strategy of Greaterp

Impart

In contrast to Anderson's fixed curriculum and top down approach to tutoring Lisp, the curriculum in Impart is not predetermined and allows for experiment, the emphasis being laid on developing general skills of programming without memorising particular detail [Elsom-Cook 1984]. The author describes the system as ideally being "regarded as a teacher watching an interaction between pupil and environment, only interrupting if it seems necessary". The environment referred to is a Lisp syntax directed editor and interpreter, previously implemented as a menu and template interface to Lisp [Elsom-Cook 1983].

This open-ended approach requires relatively sophisticated student modelling. This is achieved in a three stage process, creating a 'bounded user model', which should reflect the likely level of learning at a given stage. The student model is arrived at by successive refinement of estimates of the user's progress. Based on a lower bound of the

minimum amount a student would have learned from a particular interaction and the optimal learning which could have taken place, adjustments are made by the system using machine learning techniques, till an acceptably accurate model is reached.

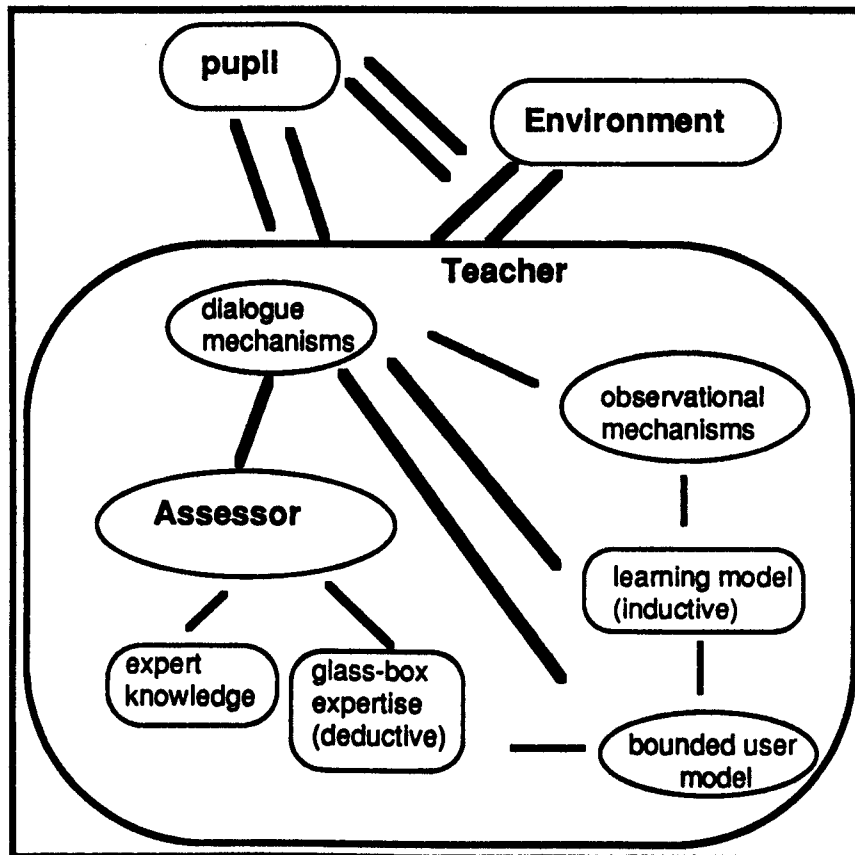


Fig.1.13. Design of Impart tutoring system

The role of the 'teacher' is envisaged as one based on the 'principles of computer coaching' discussed by Burton and Brown [1982]. The educational theory underpinning interaction between the pupil and 'teacher' is that of guided discovery. Provided with a suitable environment, a student would be left to explore it, but not be left entirely unaided or without guidance. On the one hand the student would be discouraged from pursuing unfruitful lines of learning and on the other would be encouraged to progress to new topics if the student model indicated that the pupil's current state of learning was a suitable

basis for doing so. This is somewhat akin to Goldstein's 'new knowledge frontiers' principle for topic selection incorporated in his design for a tutoring system [Goldstein 1982]. The diagram above (fig.1.13) gives some idea of the architectural design of the system.

Trill

The Rather Intelligent Little Lisper [Cerri, Elsom-Cook & Leoncini 1988], is a system designed specifically to perceive and correct misconceptions which may underlie novice Lisp programmers mistakes. Based on a semantic network of types of concepts and types of relations, the progress of the student is tagged, following a semantic path of text given, text satisfactorily answered and text-knowledge verification. In practice this means that should the student make a mistake in answering the given questions about a Lisp concept, the system follows its concept-analysis path back through the internal subconcepts. At the juncture on this concept-analysis path at which the student shows by correct answering of questions that she has understood the material presented, the system then begins to work its way back "up" the path to the original concept being taught. If, for instance, given the following question :

**"What is (CDR L) when L is the argument
(SWING (LOW SWEET) CHERRY) ?"**

the student failed to perform the operation of removing an element from the head of a list and returning the tail correctly, i.e.

((LOW-SWEET) CHERRY)

then the system would begin a series of checks to discover whether or not the student understands the concepts involved i.e. the 'head of a list', or a 'list'. A "rock bottom" situation where there are no more relevant subconcepts to present, triggers explanatory text about the last

topic, after which the system attempts to lead the student back "up" through the subconcepts again.

The teaching strategy of Trill is strictly socratic, with similarities to the Scholar tutoring system [Carbonell 1970], the student model being basically an overlay of the "expert" semantic net of Lisp concepts. It has relatively limited aims and has not yet been extended to include more than a limited subset of Lisp. It is interesting however in that it attempts to tackle possible misunderstandings of basic concepts, a problem that must be addressed in any computer tutoring system. Such misunderstandings can be the cause of errors at a higher level and the earlier they are diagnosed the better.

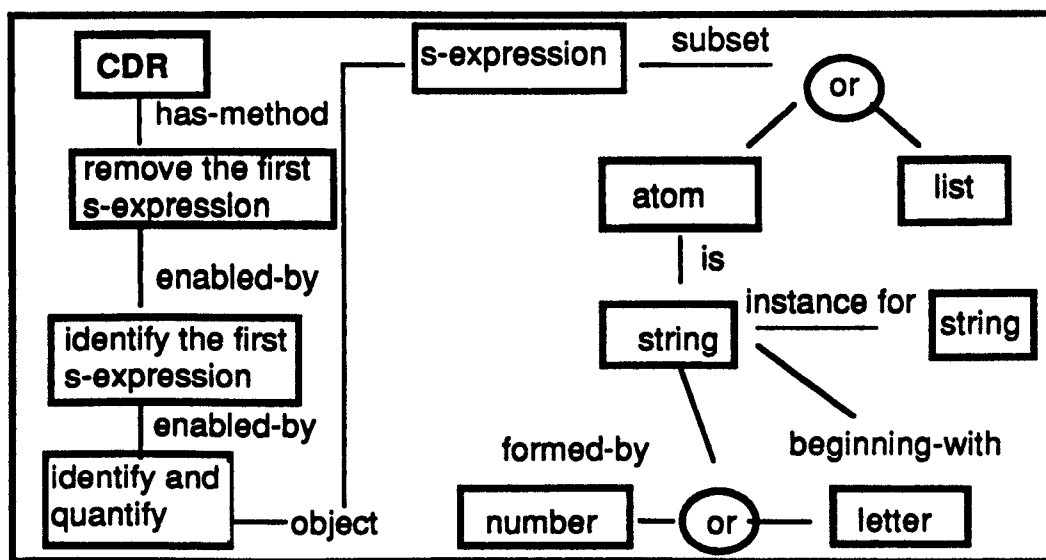


Fig.1.14. Section of Trill's semantic network

As can be seen from the above diagram (fig.1.14), the system has links from the top-level operation, in this case finding the cdr of a list, to the concepts involved in carrying out that operation. Here this entails knowing what the head of a list (first s-expression) is and ultimately knowing what a list is.

1.3.2. Pascal Tutors

Two systems designed to tutor Pascal are described briefly below. One system, 'Proust' [Johnson & Soloway 1985] is designed to help students who have sufficient programming experience to produce a syntactically correct program on their own, while the other system, 'Bridge' [Bonar & Cunningham 1986] does not presuppose any programming experience whatsoever.

Proust

Although Proust (Program Understanding for Students) as it stands does not call itself a programming tutor, it is currently being designed as such [Johnson 1986] and must be considered in the same class, since its purpose is to identify semantic errors in novices' programs and to give them advice and instruction where it judges these errors are symptomatic of underlying misconceptions.

Novice programmers complete their programs and if the programs are compiled without errors being signalled, they are then passed to the Proust system and undergo 'intention-based analysis' before being executed. The system uses a library of stored descriptions of the programming problems to be solved by the students, a library of programming plans or 'cliches' which are considered necessary to solve these problems and a library of 'buggy' plans, based on errors gathered from the work of novice programmers.

The system 'knows' the intentions of the programmer, because it is following a formal problem description of the programmers problem, reduced to the goals and subgoals which are necessary to achieve the solution.

Analysis is based on program synthesis; the system working through the goals of the problem description, matching them to plans likely to be used to achieve these goals, then trying to match these plans against the code used by the student. If no correct plan approximates code in the student program, then a match is sought in the library of 'buggy' plans. If an approximate match is found then the suspected error is stored and when the analysis is completed, the student is given the results and appropriate advice.

Although in some twenty percent of cases the system cannot diagnose with certainty the error and probable cause, its performance seems to indicate that it is effective within the range of programs that it can handle. An eighty percent success rate [Johnson & Soloway 1985] in giving novices on-line help and advice about the probable source of their errors, possibly at the level of underlying misconceptions of the programming language, seems impressive.

It must be borne in mind though that at present only a limited number of programs can be used, within a subset of Pascal and its analysis by synthesis method involves an extremely high overhead of stored information.

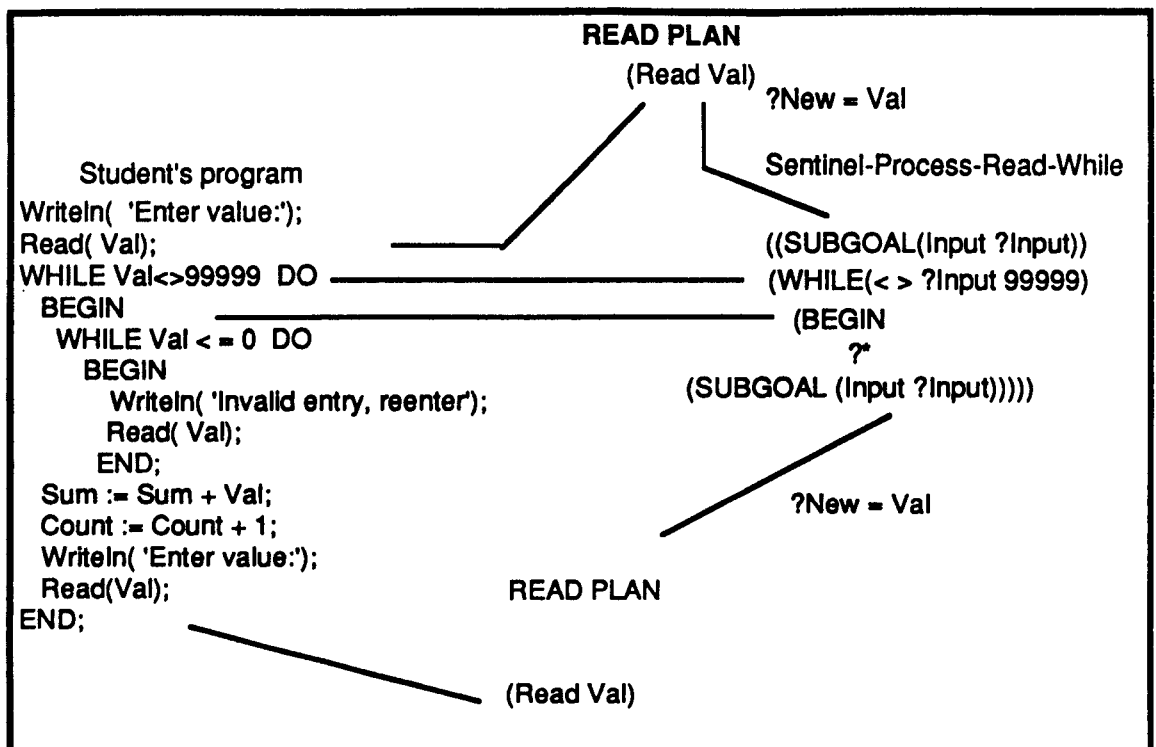


Fig.1.15. Matching process in Proust substituting in known values e.g. 99999 is filled in as the value of Sentinel

Bridge

In contrast to Proust, the Bridge system [Bonar & Cunningham 1986], currently being developed as a commercial software product, is designed to help novices from the very beginning of their programming experience. One of the main aims of the authors of this system is to address the problems novices experience in the program planning stage. The system helps the pupil to write a complete program solving a predetermined problem. This is done by helping the student progress through successive stages of approximation from a natural language solution of the program problem to the complete coded program.

The system is multi-windowed, menu based, and breaks down the programming process into three stages. Initially the student is given the problem to be solved, displayed in a window on the left of the screen. A window in the middle of the screen lists the sort of natural language

phrases that the student might use in producing an algorithm. The pupil constructs an algorithm from these in a third window on the right of the screen. In the next phase this algorithm is translated into programming 'plans', again by matching its components to a given choice of programming plans, and in the final stage these plans are matched to programming constructs and used to build the final program. Templates are provided for the programming constructs, appropriate advice is displayed in an upper window and at any point 'hints' can be chosen from the menu display. The hint facility bases its response on the results of an analysis of the student's work to date. This analysis is carried out by comparing prestored plan requirements i.e. what phrases should appear in the program and in what order, against the student's solution, not unlike the basis of Proust's analysis by synthesis method.

The system is an attractive one. The use of multi-windows allows the student to see the development of the programming process and make connections between the natural language solution, an algorithm, programming plans and programming constructs. This process of working from level to level in the construction of a program is very important [Taylor 1987] and the menu selection principle provides a loosely structured framework within which the beginner is unlikely to go too far astray. It is not clear how far this visual representation of the planning process can be taken before the amount of information displayed on the screen becomes overwhelming. Monitoring the progress of the pupil relies heavily on prestored information, and at present makes it very problem specific. The flexibility of choosing from a selection of language constructs at each stage, be it natural language or programming plans, is apparent rather than real, since these must be

constrained to a selection that the system has prestored information about and can operate upon. Given these points, the system provides what its authors intended to offer to the novice, a system of learning to program which 'bridges' the gap between the novice 'line by line' approach to programming and the expert's higher level view of programming as a series of plans to accomplish the necessary goals [Jeffries 1982]. It is also a compromise between the inflexible step by step approach of Greaterp and that of allowing students the freedom of completing a program without on-line tutoring help, with the attendant risk of the results being then too 'bug-ridden' for successful analysis.

1.3.3. Help in Fortran

The system 'Laura' [Adam & Laurent 1980] is, like Proust, called by its authors 'a debugger to teach programming' rather than a tutor, but similarly to Proust, it is designed to tutor novices. It takes as input a syntactically correct program and analyses it in order to detect semantic errors. Unlike Proust, it is not based on 'best' matching between a set of stored goals, programming plans, buggy plans and the student's program. The system has a prestored 'program model' which represents the 'correct' programming solution and this is compared with the student program. If the two do not match, both the student program and the model program undergo a series of transformations in order to reduce them to representations of their functionality. If the resulting transformations can be shown to be similar, the program is judged correct, if not, then the areas of dissimilarity are suspected to be symptomatic of an error or errors in the student's program. Like Proust, it allows the student to complete a program before getting on-

line help, but the student is constrained to producing a program which functions in the same way as the 'expert' model.

1.4. Tutoring for Prolog

Having looked at the tutoring systems developed for other languages, it is interesting to consider which features of these tutoring systems would be of use or relevant to tutoring Prolog novices, what lessons could be learnt from their experience and what directions work on a tutoring system for Prolog should take. The first point that is obvious from the above section, is that there can be no one 'tutor' for Prolog. Each of the systems described above was created and designed to fulfil a need for a certain sort of tutoring at a certain stage of learning. To a certain extent the significant differences between these systems reflect different perceptions of novices' needs. These needs vary from help with the most basic concepts of the language and seemingly trivial details to quite sophisticated 'bug' detection to explain why their first programs don't work as expected. At times they need guided, even structured learning, at others they must have opportunities to make mistakes, but must also receive assistance so that they can learn from their mistakes. A Prolog tutor must be thought of as an integrated system comprising a variety of modules designed to meet these different needs. This being so, then it must also be an intelligent system since it must have some sensitivity to the current needs of the user. This suggests inclusion of a component equivalent to the design of the 'teacher' element in the Lisp tutor *Impart*, incorporating a dynamic model of student behaviour established by modelling processes similar perhaps to those described by Elsom-Cook [1984], which in combination with direct pupil-interaction could determine the module currently best suited to the student.

1.4.1. Tutoring modules - PITS

Consideration of which kind of modules would be incorporated must be dictated by the different levels of learning at which novice programmers need help and by what sort of help they need at those levels. A proposed module designed for inclusion in the construction of an intelligent tutoring system for Prolog, PITS [Looi & Ross 1987], [Looi 1988] has been developed and tested at Edinburgh. It is intended for students who have learnt the basics of Prolog syntax and semantics and have a reasonable grasp of Prolog's control flow [Looi and Ross 1986]. It is envisaged that a novice could use a module such as this within the framework of an intelligent tutoring system, to explore and experiment with constructing programs, getting almost immediate feedback about code that she has input. The automatic-debugger 'Apropos' currently deals with a small class of programs involving recursive list processing. Using a combination of analyses to check the student's program, it attempts to recognise standard techniques used in recursion, tutoring the student on any data-flow anomalies, type errors, missing base cases or certain classes of typing errors. Reporting on results using actual students' solutions, the authors found that for most students the message returned when an error was discovered enabled the users to correct their work successfully. This represents a useful contribution to the current research into Prolog 'techniques' [Brna et al 1988].

1.5. The Prolog Interpreter

There is however, also a need to provide on-line tutoring help for novices at other levels. It is apparent from overviewing the problems which novices experience, that for those beginning Prolog it is extremely important that they develop reliable models of the 'virtual' machine

discussed in section [1.2.1]. Ideally, in a declarative language, errors should be able to be explained in terms of a missing case or an incorrect representation. While this logic-based declarative view of the language is often a beginner's introduction to Prolog, the interpreter is relentlessly procedural in its depth-first goal directed search. As we saw in section [1.1.5], a significant number of novices' problems can stem from a misunderstanding of this procedural behaviour of the language. Students who are not aware of the importance of clause order in Prolog execution, or are not aware that on failure the interpreter will systematically backtrack, trying to resatisfy previously satisfied goals, will find that their programs produce unexpected, often 'buggy' results. Coombs and Stell [1985] stress this distinction between what they term 'specification' errors, i.e. errors in the specification of the solution to the programming problem, and 'procedural' errors, i.e. errors due to misunderstanding of the procedural aspect of Prolog. Detection and analysis of these procedural errors would allow an understanding of the student's model of the interpreter, thus providing the basis for offering useful tutorial help.

1.6. Summary

In this chapter we have overviewed the problems which are experienced by novice programmers. A conclusion to emerge from this overview was that not only do novice Prolog programmers experience problems which are common to most students in the initial stages of learning to program, but that they also encounter difficulties which are language specific. In particular they face problems posed by the hidden procedural nature of the Prolog interpreter. We have also considered what help is currently available to novice Prolog programmers to ease

this learning process and to encourage their awareness of this powerful procedural aspect of the language.

Current research indicates that in the not too distant future there are likely to be significant improvements in Prolog environments, particularly with the advent of graphical debugging. This will not however necessarily address their problems in understanding the underlying mechanisms of the Prolog interpreter.

The inclusion of a trace package such as the Transparent Prolog Machine [Eisenstadt & Brayshaw 1988] within an on-line Prolog tutoring system, would almost certainly help students to develop a valid model of the Prolog interpreter right from the start, thus helping to eliminate 'procedural' errors due to a misunderstanding of the Prolog interpreter.

However, there would, on the one hand, always be students who would need extra help in interpreting this model and on the other, students who would develop models of their own, which may or may not be valid. As Jones [1981] quite rightly points out, we may provide models, but novices do not necessarily adopt these models. Rather, they may form their own models based on their personal conceptions, misconceptions, interpretations, misinterpretations of the language and its execution.

This underlines the need, not currently met, for on-line tutoring to help novices with problems they may have in this area. In developing a tutoring module which would provide such help, a major component must be a diagnostic component designed to identify the procedural

errors associated with students misunderstandings and misconceptions of Prolog's interpreter.

In the following chapter, after discussing the kind of misconceptions that are thought to underlie these procedural errors, we report on an empirical study which was undertaken to investigate in more detail the Prolog execution models which novices form.

Chapter Two

2. Novices' misconceptions of the Prolog interpreter

In the preceding chapter we considered the problems of novice Prolog programmers and in particular the need to understand the largely hidden procedural aspect of that language. It is essential that students form reliable models of Prolog execution in order to write, comprehend and debug programs. We stressed the need to help students form correct models of the Prolog interpreter and put forward the case for developing a tutoring module to give on-line help in cases where students may have formed incorrect or incomplete models. The student modelling element is an essential feature of such a module, since, should it be desirable to amend or enrich the models which students form, it is first necessary to appreciate those students' perceptions of the interpreter.

In this chapter we investigate those perceptions, initially by looking at novices' errors which may indicate a misunderstanding or misconception of Prolog execution and by discussing the possible underlying causes of those errors. In doing so we draw on the work of [Fung, du Boulay & Elsom-Cook 1987] which considered the possible sources of control flow errors made by novice Prolog programmers and consider the contribution their findings make to the analysis of errors in tutoring Prolog. We then go on to report on an empirical study, undertaken to investigate specific control flow errors in more detail. The purpose of this study was to provide more information on models of the interpreter which novices do form as opposed to the models which we would like them to form. We conclude the chapter by outlining the significance of the study results in relation to addressing

the problem of automatically diagnosing faulty models of the Prolog interpreter.

2.1. A proposed taxonomy

The work of [Fung, du Boulay, & Elsom-Cook 1987] has as its aim the construction of a framework for empirical studies of novices learning Prolog. In that work we attempt to establish an initial taxonomy of 'control flow misunderstanding' errors which could be used as a basis for classifying novices' misconceptions of the Prolog interpreter and could also serve as a guide in designing diagnostic tutoring aids. As is indicated in that initial study it is clear that the area deserves closer investigation and could usefully be extended in further research. A series of empirical studies designed within the framework outlined in that study should serve to confirm and/or clarify the areas of Prolog control flow which give rise to difficulties for novices. Results may show our initial classification of students' difficulties into the different hierarchies of misconceptions discussed below, i.e. misunderstandings of the search, unification and cut processes, to be a valid one. On the other hand results from such empirical studies could indicate that this hierarchical structure, which we have seen as underlying Prolog students' misconceptions, needs to be revised. The distinction we have made in our initial study, between search errors and unification errors, for example, may be less clear cut than we have implied. This would indicate that the framework outlined in our initial study needs to be revised and would lead us to adjust our taxonomy accordingly. In this section we briefly overview the ideas of that initial study and indicate the implications of it for the research reported in this thesis.

The discussion of possible control flow errors draws on two sources, previous empirical work in this field and observations made by those involved in the teaching of Prolog to novices. As Prolog is a relatively new language, a sizable body of established empirical work with novices is not yet available, but this work can be seen as building on and extending that of [Coombs & Stell 1985], [Brna & Pain 1985], [van Someren 1985] and [Taylor 1987]. In it we see control flow errors as falling into three main categories, those which seem to be related to misunderstanding of the search and backtracking process in Prolog, those which seem to have their source in an incomplete or faulty knowledge of Prolog's method of unification and a class of errors which seem to arise from confusion over the effect of the cut operator. In sections 2.2.1 through to 2.2.3 we give a small selection of the errors which are discussed in that work and the misconceptions which are considered as the underlying causes of these errors. These are drawn from each of the categories listed above, though as will subsequently be discussed, in the research work reported here the system being developed deals only with those in the first category, i.e. misconceptions arising from a misunderstanding of the search and backtracking processes in Prolog. We show below the two types of diagram used in the proposed taxonomy to illustrate correct and incorrect models of Prolog execution and notes on their interpretation. The aim of the first type is to make the execution behaviour of the program in relation to the database explicit, by using arrows to show the correspondence between subgoals and their satisfying relations in the database. The diagram below for example would be interpreted as indicating that given the program:

p if a & b.
a.
b.

and the goal of proving 'p' the Prolog interpreter would try to prove the subgoal by matching it with the subgoal 'a' and then go on to prove the subgoal 'b'. When both subgoals have been matched successfully the program has succeeded since 'p' is proved true.

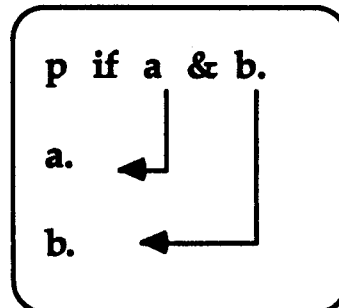
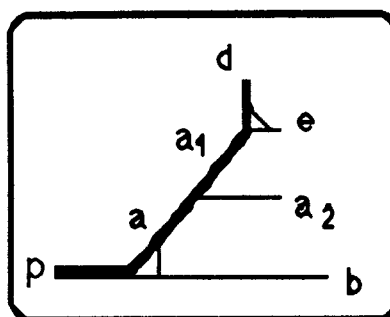


Fig.2.1. arrow diagram (taken from Fung et al [1987])

The second type of diagram shows the execution space of a given goal by the interpreter: either the actual interpreter, or the actions of the interpreter hypothesised in the novice's misconception. Disjunctions are shown as separate branches and conjunctive branches are linked by lines.



Given the program:

p if a & b.

a if d & e.

b.

a.

and the goal p, this diagram illustrates the program after the call of p to the subgoal a and then to its subgoal d. The heavier dark line indicates the flow of control up to that point and should be seen as flowing from left to right. The two a clauses have been subscripted for clarity.

Fig.2.2. Control flow snapshot (taken from Fung et al [1987])

2.1.1. Errors related to Prolog search and backtracking

In the search pattern of the Prolog interpreter, goal queries are proved to be true or otherwise by a process of unification and resolution e.g. Robinson [1965]. The order of resolution implemented by the Prolog interpreter is strictly linear and depth-first. This determines the way in

which the interpreter searches systematically left to right across and top to bottom down through the database in an attempt to satisfy the current goal. A goal failure instigates strictly chronological backtracking, the interpreter returning to the most recently proved goal to seek an alternative proof. If this is done successfully, the search continues again in a forward direction, if not, then backtracking continues to earlier goals. In the case of no earlier goals being able to be resatisfied, the program fails. The following errors are a few of those seen by Fung et al [1987] to be related to misconceptions concerning this search process.

Facts before rules

As example of this error, in the following program, when attempting to satisfy the goal *p*, students expect Prolog to choose the fact *a₂* in preference to the rule *a₁ if d & e & f*.

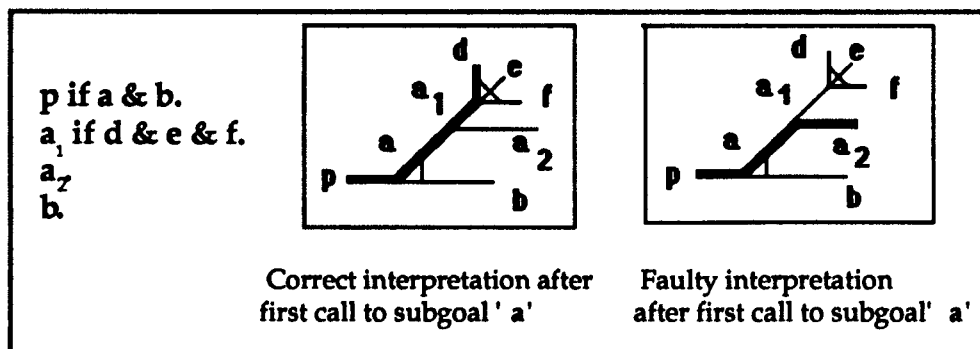


Fig.2.3. facts before rules

The misconception that the interpreter scans facts before rules, which gives rise to this error, may come about because facts are typically used as stopping conditions for recursive calls and consequently are often deliberately placed earlier in the database. The vocabulary used in teaching Prolog does also in many courses tend to stress the difference between 'facts' and 'rules', perhaps leading students to believe that

Prolog itself chooses to scan facts before rules, rather than being dependent on the choice made by the programmer. A variant on this, again possibly due to attaching undue importance to the format of programs and the vocabulary used in teaching novices, or from faulty generalisation from examples [Brna personal communication], is that, similarly, the Prolog interpreter distinguishes between rules and facts, but in this case the belief is that it scans rules before facts. In the program below this would lead to a situation where the student tries to match the subgoal *a* with the rule *a2* before trying to match it with the fact *a1*. Again the goal is *p*.

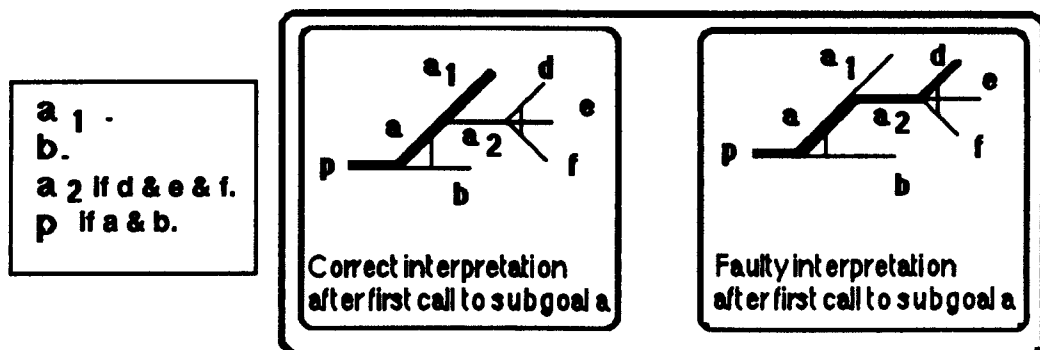


Fig.2.4. rules before facts

Try once and pass

Here the student fails the parent goal immediately after the first failure of any subgoal that is encountered. This reflects an incomplete model of the interpreter in which the student is unaware of the exhaustive backtracking which takes place when a subgoal fails. It has also previously been noted by Coombs & Stell [1985]. In the following program for instance, given goal *p*, the correct interpretation would be:

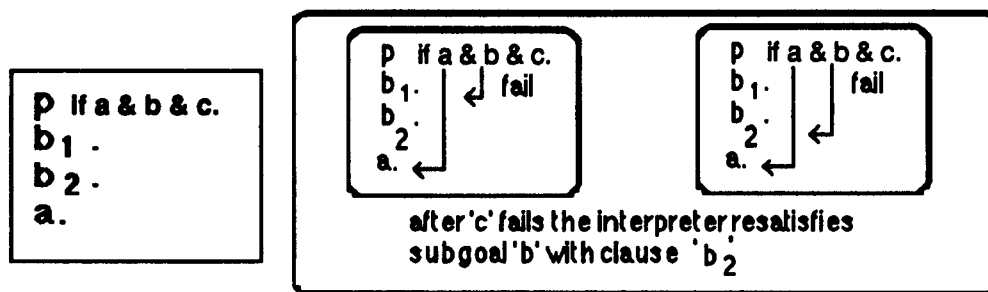


Fig.2.5 correct interpretation

However, a novice with a 'try once and pass' model of the interpreter, although correctly predicting the failure of *p*, predicts it at too early a stage in the program's behaviour, as shown below:

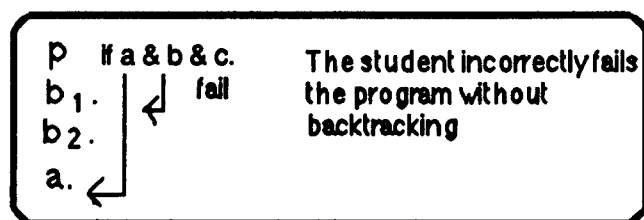


Fig.2.6 incorrect interpretation

Redo from left

Again, previously noted by Coombs & Stell [1985], this is a misconception about the behaviour of the Prolog interpreter in the backtracking process, an example of which is given below. Given a query *p* to the program:

```
p if a & b & c.
b1.
b2.
a.
```

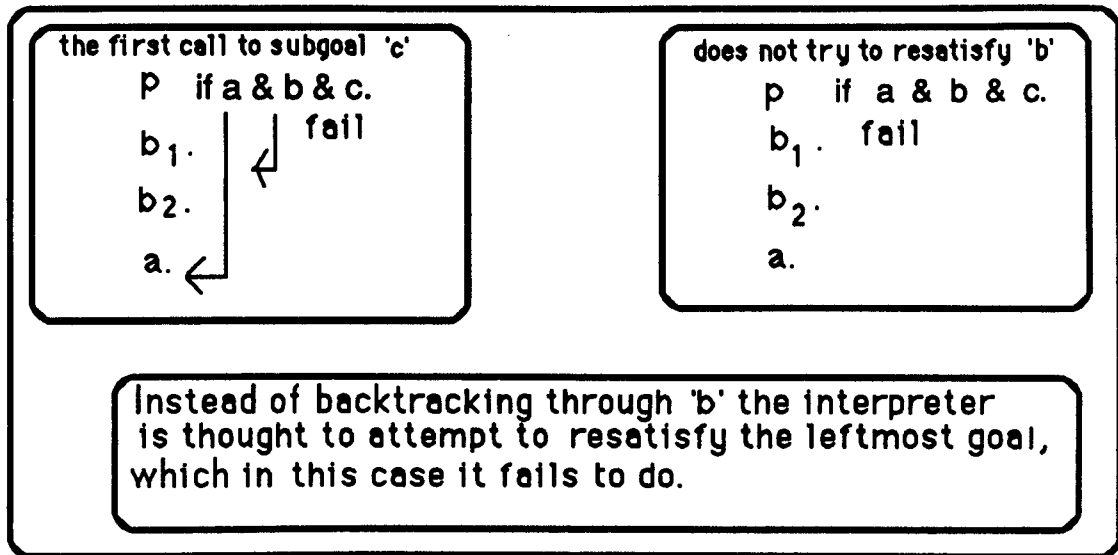


Fig.2.7 redo from left

the student imagines that on the failure of the subgoal *c*, the flow of control returns immediately to try and resatisfy the leftmost subgoal, rather than backtracking chronologically to reprove subgoal *b*. The outcome of the query is correct, but it is based upon a faulty model of the interpreter.

2.1.2. Incomplete or faulty knowledge of unification.

The term unify is used here, although strictly speaking it is not identical to the unification used in resolution [Clocksin and Mellish 1984]. There are only three essential principles:

- two terms unify if they are identical.
- if the terms contain variables, they unify, if when the variables become instantiated, they then become identical.
- a variable can be unified with any structure.

Nevertheless unification manages to give rise to errors, of which the following two are examples.

One value can only unify with one variable

This misunderstanding entails the belief that only one variable may unify with a particular value [van Someren, 1988]. This misconception arises perhaps from a failure to appreciate that a variable is simply a

value holder. It is possible to store a particular value under several different 'addresses', much as the same Joe-Bloggs may turn out to be the value of 'Owner-of-Rose-Cottage', the value of 'Owner-of-Knightsbridge-Pad' and also the value of 'Owner-of-Fido'. In predicting the backtracking or failure of unification, this buggy belief that two variables may not instantiate to the same value would cause the following query to fail

?- g ([A,B]) = g ([2, 2]).

Fig.2.8. Can two variables have the same value?

whereas the correct prediction is that the variables 'A' and 'B' will succeed on assuming identical values, since both unify with the integer two.

false interpretation
 ?- g ([A,B]) = g ([2, 2]).
 no

correct interpretation
 ?- g ([A,B]) = g ([2, 2]).
 A = 2

Fig.2.9. the correct answer

One variable can unify with many values

In Prolog a particular variable may be correctly assigned only one value within the predicate in which it occurs. A call to retrieve the value stored under a particular variable name will always produce only that one value. The position of the variable and its repeated use in a clause does not alter this. A student misunderstanding this may believe that the same variable can simultaneously hold two different values, or that the same variable name repeated within a predicate may represent different values. This results in the belief that a call to that variable may sometimes return one of these values, while at other times the

second value may be returned. A query, in order to test the success or failure of unifying the following terms,

?- $f(pat, igie) = f(Z, Z).$

would be incorrectly predicted as resulting in success with the instantiations

$Z = igie$

$Z = pat$

2.1.3. Confusion over the effect of the cut operator

The cut is an inbuilt operator which can be used to limit the Prolog backtracking process. When invoked, it always succeeds. In doing so it effectively 'freezes' the values of any variables which have been bound in the subgoals of the predicate in which it is invoked, the parent goal, and any further untried clauses of that parent goal are removed from the search space. The diagram below shows a program in which the cut is used.

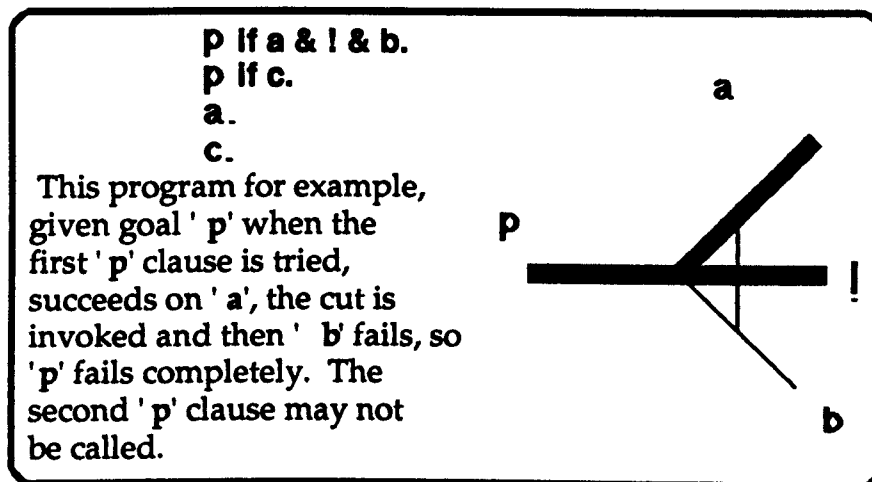


Fig.2.10. correct interpretation of the scope of the cut operator

The two following examples however, illustrate the sort of confusion which arises for students who have trouble forming a correct model of

the cut operator, the first under-estimating , the second over-estimating its scope.

The cut freezes the clause but not the parent goal

Given the following program and the query p,

p1 if a & ! & b.

p2 if a.

a.

in a correct interpretation,

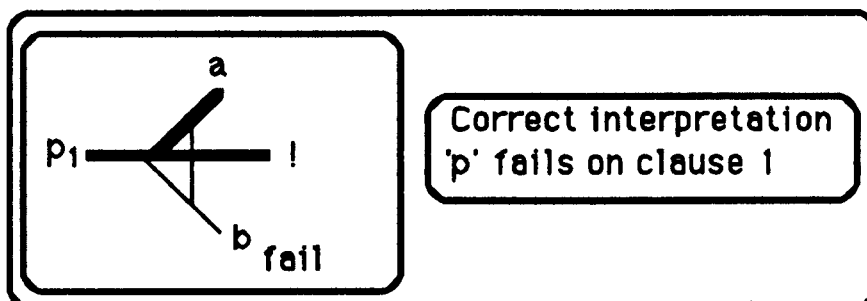


Fig.2.11. correct scoping of the cut operator

In a faulty interpretation, rather than regarding the cut as freezing the current predicate and its clauses, the student may assume that the cut has the effect of freezing only the particular clause in which it occurs , so assumes that after p1 fails, the second clause of the parent goal p may then be tried:

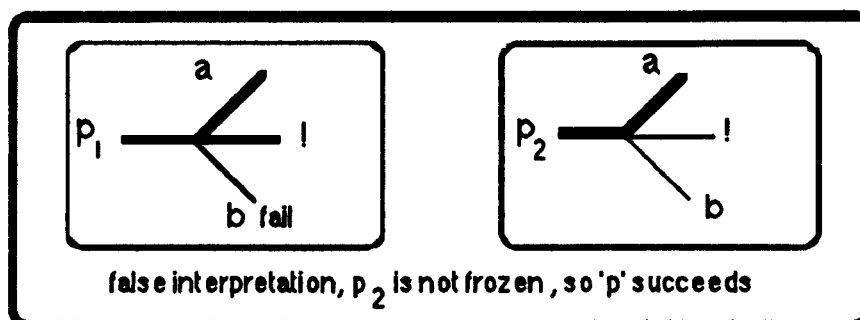


Fig.2.12. under-estimation of the scope of the cut operator

The cut freezes the grandparent goal

In this faulty model of the interpreter, instead of the cut immediately freezing the parent goal of the clause in which it occurs, the student

perceives it as freezing the clause which invoked this parent goal, the 'grandparent', e.g. in the following program and given the query d ,

```

d1  if    p(1).
d2  if    p(2).
p(X) if   a(X) & ! & b(X).
a(1).
a(2).
b(2).

```

where a correct interpretation would be as shown on below (fig.2.13).

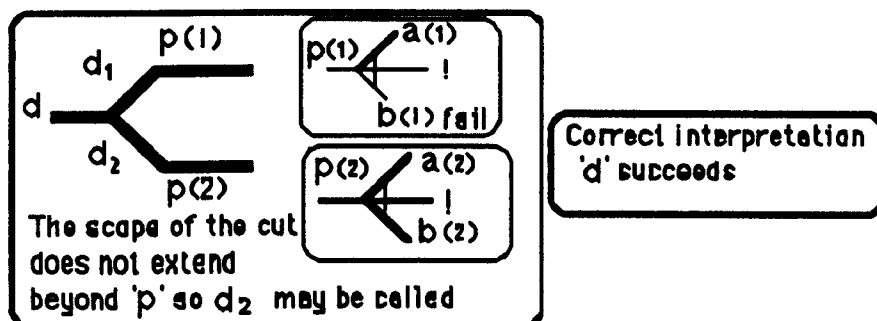


Fig.2.13. correct scoping of the cut operator

in a faulty interpretation the student has seen the scope of the cut as extending beyond its own parent goal p to the parent goal of p , i.e. d .

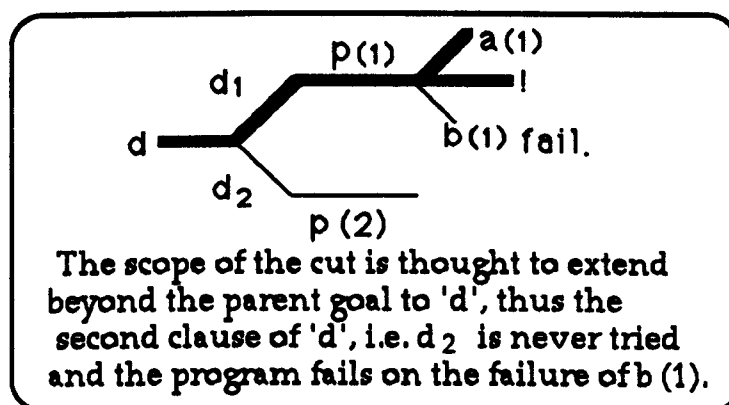


Fig.2.14. over-estimation of the scope of the cut operator

2.1.4. Implications

The approach taken in Fung et al [1987] is of twofold significance. In this discussion of novices' control flow errors, an attempt is made to clarify the possible sources of novices' misconceptions and

misunderstandings of program execution which lead to those errors. The errors reported in that work, of which a selection were shown above, are drawn from the small amount of established empirical work available and from observations of novices' work, although that work did not form part of a controlled study. The investigation of possible misconceptions and misunderstandings of Prolog control flow showed that these errors could stem from several sources, ranging from the manner in which Prolog is taught to the attitudes and experience which the learners bring to the task of programming. In attempting to relate the error to the underlying source of that error, this work is a step towards the goal of a more meaningful analysis of errors in Prolog tutoring. It is also significant in that it highlights the range of difficulties which students have in appreciating the underlying mechanisms of the Prolog interpreter and the need for planned empirical research to be undertaken in order to increase our knowledge of these problems. The implications of that work for the research reported in this thesis are that if we are to tutor Prolog novices effectively, we must undertake the necessary empirical work to examine more closely the areas which cause difficulties and produce errors. Having done so, the emphasis must lie on treating the underlying misunderstandings which cause these errors. The following section of this chapter reports on such a study, which focused on one particular area of Prolog control flow. It was designed to elicit novice Prolog programmers' predictions of the backtracking process, the intention being to conduct the experiment in a way which would reflect as nearly as possible the subjects' models of control flow. The results were expected to be of interest in terms of gaining insight, firstly, into the level of difficulty which novices experience in understanding this aspect of control flow and secondly, into the underlying misconceptions

which lead to incorrect predictions. The results would also establish an empirical base which would be of use in the wider goal of developing a diagnostic module for modelling and identifying students' misconceptions of the interpreter.

2.2. An empirical study

The students who took part in the experiment did so on a voluntary basis and out of fifty possible subjects, thirty-six completed the questionnaire. The analysis of the results of this experiment is based on data from thirty-two of these students (four students had received help while completing the questionnaire, so this data was omitted from the results). The students were approximately the equivalent of third year psychology students (an exact equivalence is not possible, since these students were distance learners taking an Open University degree course). As part of their course, the students were attending a summer school week at Sussex University, during which time they completed a psychology project which involved programming in Prolog. The project took about two days, during which they were able at any time to ask advice and help from the course tutor. Their task consisted of designing an algorithm which attempted to model a cognitive process. The students were then required to write, run and debug a short Prolog program which implemented this algorithm.

Prior to the summer school week all these students had studied a short preparatory book-based introduction to Prolog [Eisenstadt 1987]. This introductory booklet covers the basic concepts of Prolog, i.e. facts and queries, the query interpreter, conjunctive queries, rules, database search. All students were expected to have completed the book-based

course and accompanying exercises, but some had not had any hands-on experience of Prolog programming before their summer school work. Others had some experience of programming but in a language or languages other than Prolog, while a few had some hands-on experience of Prolog and of other languages. Figure 2.15 overleaf shows the relatively small percentage of the subjects who had prior experience of programming both in Prolog and one or more other programming languages.

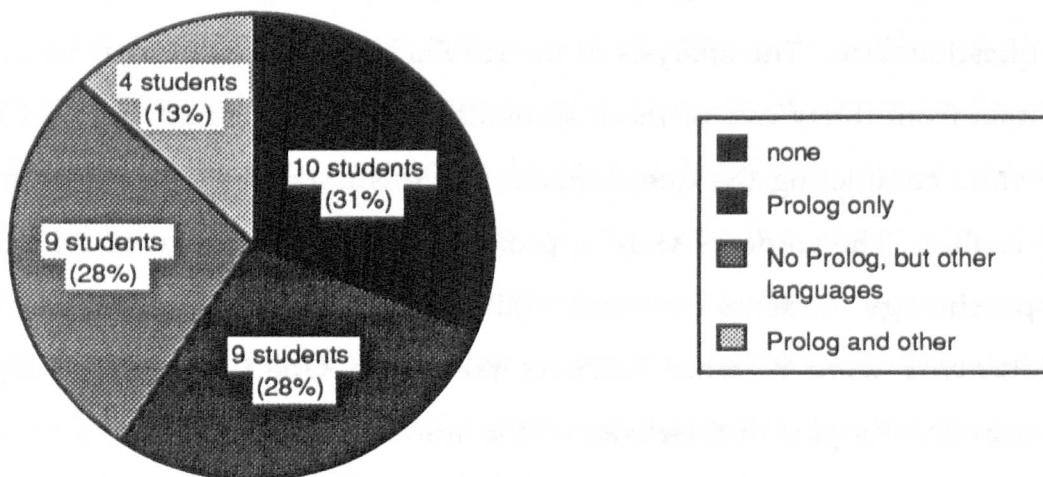


Fig.2.15. Programming experience prior to the O. U.course.

2.2.1. The experiment

In the experiment each subject was given six Prolog programs and asked to describe what she believed would be the action of the Prolog interpreter in each program. Of the six Prolog programs in the questionnaire given to the subjects, in five, a query to p , in a correct model of Prolog search, would instigate backtracking due to failure of a subgoal. A sixth program was expected to produce a correct prediction in every case. i.e.

p if a .
 a .

The latter was included in order to give all the students the opportunity of feeling confident about at least one of the programs and as an example of a program in which no backtracking normally takes place. A sample copy of the questionnaire which was given to each subject is included in appendix A1. For the purposes of the experiment a variable-free subset of Prolog was used. This was to restrict the area of errors which could be made to those involving the order of search and backtracking of the Prolog interpreter and to narrow the range of possible interpretations of students' errors. The programs were chosen to be as simple as possible while allowing the students' answers to reflect their belief about the order in which a Prolog program is executed. For example, in the following program:

```
a.
b.
b.
p if a & b & c.
```

given the query *p*, a correct prediction of the Prolog's search would be to try and prove *p*. To succeed in doing this, the interpreter tries to prove *a*, succeeds, then tries to prove *b*, succeeds, then tries to prove *c*. On failing to prove *c*, the interpreter would backtrack and try to resatisfy *b*. In this case, the goal *b* can be resatisfied, leading to another attempt to prove *c*.

In the notation used in the experiment this would be represented as:

p try	a try	a succeed	b try	b succeed	c try	c fai	b try
b succeed	c try						

A student may, however, predict that after the first failure to prove *c*, the interpreter returns to the top of the database and tries to resatisfy *a*, which would be recorded in the way shown below:

p try	a try	a succeed	b try	b succeed	c try	c fail	a try
----------	----------	--------------	----------	--------------	----------	-----------	----------

If this faulty pattern of backtracking is consistently predicted in similar problems, then it is reasonable to assume that the student has formed a particular model of the Prolog interpreter, in this case an incorrect one, the 'redo-from-left' discussed earlier. By following the student's step-by-step prediction of program behaviour in a selection of simple problems, any consistently faulty or incomplete model of Prolog's execution which has been formed by the student should become apparent.

The experiment was carried out as a paper-and-pencil exercise. The students were told that they were not expected to spend more than half an hour over their answers, and might well complete them in less time than this. They were asked to work individually rather than collaboratively and it was explained in the questionnaire that if they felt unsure about their answers, then they were free to consult a tutor after they had first completed the questionnaire. They used the notation shown in the preceding paragraph in order to record their predictions of the steps the Prolog interpreter would take to prove each query. An explanation of the notation and an example program and answer were given at the beginning of the questionnaire. If requested, the experimenter also presented this example page verbally when giving the booklet to the subject.

The results of the experiment were analysed as follows. The prediction of the interpreter's action given by each student to each problem was compared with a 'correct' prediction of the interpreter's behaviour. Where a difference was found, the student's answer was then compared with the answer which would have been produced if the student had

given her or his prediction based upon one of the hypothesized faulty models of the interpreter described below. If the student's prediction fitted the pattern produced using one of these faulty models, it was noted as an error of that category, otherwise it was noted as an error of an unidentifiable sort.

2.2.2. Problem design

In designing the problems set in the experiment, an attempt was made to produce a range of simple programs which would allow certain expected misconceptions to be apparent. There were five error types felt to be representative of novices' misconceptions of the interpreter and which we were expecting to find in the subjects' answers. Three of these have been described earlier in this chapter, i.e. 'facts before rules', 'try once and pass' and 'redo from left'. Of the remaining two, one is briefly described below, while a description of the second, 'redo from left preserving pointers', forms part of a subsequent discussion on the subject of multiple errors.

One pointer per clause

In normal Prolog search, when a goal or subgoal has been satisfied i.e. matched successfully against a fact or rule in the database, this match is recorded by the Prolog interpreter in case an attempt to resatisfy the goal must be made at a later point in the program. If this happens and the goal in question cannot be resatisfied, then it fails and any 'match' in the database is 'forgotten'. Consequently, if a fresh call to that goal is made, a match can again be made in the database. In discussing this error [Fung et al 1987] we suggest that this misconception may be related to the method of teaching the order of Prolog search. This is often taught in terms of a 'marker' or 'pointer' which the interpreter places

in the database as program execution proceeds. A student suffering from the 'one pointer per clause' misconception imagines that once a goal or subgoal has been matched against a fact or rule in the database, then the 'marker' placed by that fact or rule remains there for any subsequent execution of that clause, disallowing its use in any further goal calls which may legitimately be made. A possible misunderstanding is to believe that only a single marker is available for each clause and that as various goals are marked this pointer moves monotonically down through the database, so making it impossible to satisfy certain goals if the "marker" has passed the clause in question. In, for example the program below and given the goal p .

p_1 if $a \ \& \ b \ \& \ c$.

p_2 if $a \ \& \ b$.

a_1 .

a_2 .

b .

in a correct interpretation, the first clause, p_1 eventually fails because there is no definition of the relation c in the database, but the second clause, p_2 succeeds.

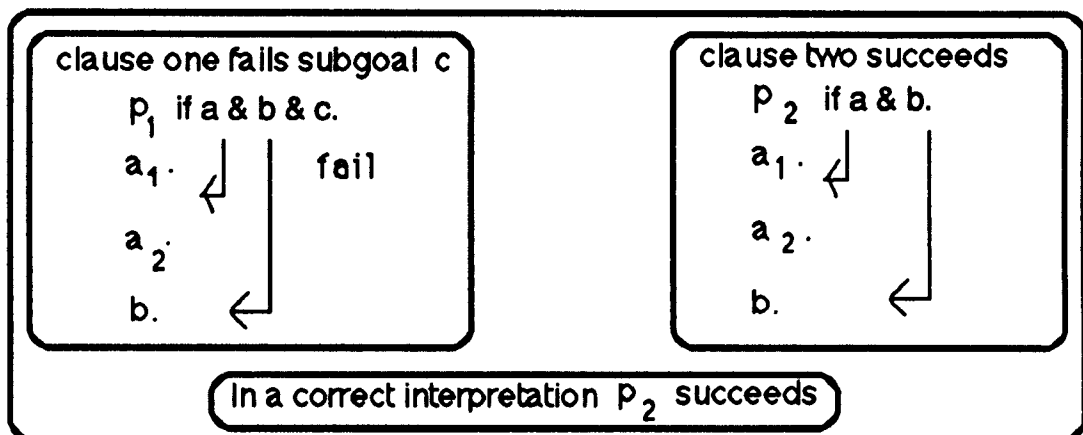


Fig.2.16. correct interpretation

In an incorrect interpretation, the attempt to satisfy subgoal c in order to prove p_1 moves the "marker" for an a clause and a b clause down past the first clause of each.

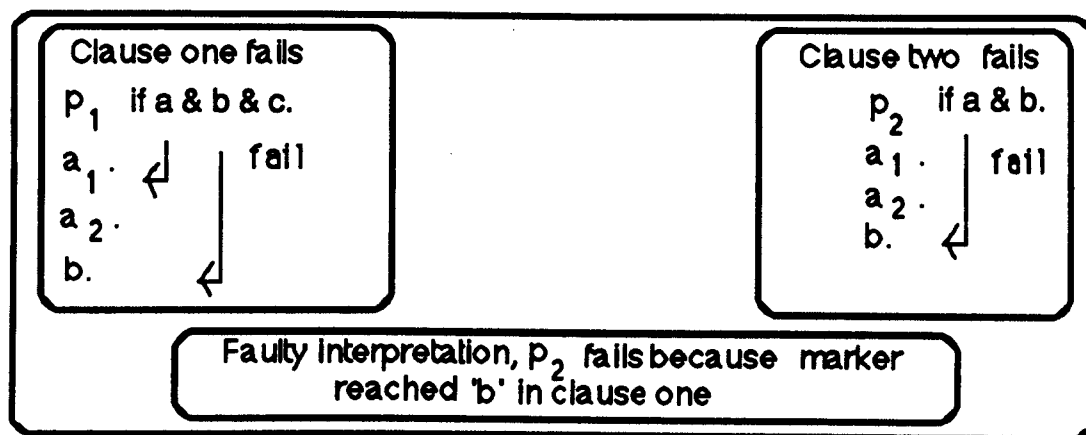


Fig2.17. incorrect interpretation

In the attempt to then succeed in p_2 , the subgoal a is satisfied trivially by the fact a_2 , but the subgoal b fails, because its marker has moved past it in the attempt to succeed in p_1 , so p_2 fails.

Multiple bugs

The question of multiple bugs is one that is frequently raised when error analysis is under discussion [Burton 1982], [VanLehn 1983] and is occasionally put forward as grounds for invalidating the 'error analysis' approach to on-line tutoring. It is perhaps worth taking a quick look at the area and putting the question into perspective, since if we were to accept that conclusion, we run the risk of throwing the baby out with the bathwater. The use of error analysis to determine the tutoring needs of students is not by any means a new concept. This approach was the basis of work by [Brown and Burton 1978] and [Young & O'Shea 1981] in the domain of arithmetic. A realistic gauge of a student having mastered a skill is when we can see that a task which demands the application of that skill has consistently been successfully completed. When, on the other hand, a student consistently fails to complete a task requiring that skill, then there are reasonable grounds for believing that the skill has not been mastered. Depending upon the nature of the task, an analysis of the errors which an unsuccessful student makes can give

valuable insight into which aspect of the skill is giving rise to difficulties. The results of this analysis can then be used to provide suitable remedial help. In both the works referenced above, the detection and analysis of errors in the subtraction process was used to diagnose students' misapplications of the skills and sub-skills involved. We have seen in the previous section that an analysis of errors made by students in predicting the control flow of simple Prolog programs can indicate what misconceptions they harbour concerning the action of the Prolog interpreter. Detection of these errors in students' work would then indicate the area(s) in which help is needed and facilitate the task of offering tutoring which is relevant to their difficulties.

So far so good, but then come the awkward questions. How do we define consistency in this context? How many times must a student produce a correct solution before it can be presumed that the relevant skill has been mastered? How can errors be distinguished from slips caused through temporary lack of concentration [Norman 1981]. How do we separate out overlapping errors, or 'multiple bugs'? As has been pointed out in other studies of errors [Burton 1982] their classification is made more complex by this possibility of combinations of particular 'bugs'. It is not difficult to imagine a situation in which a novice programmer might produce an interpretation of a program which displays symptoms of more than one misconception. How do we deal with apparently haphazard errors? It is a possibility that a novice displaying the symptoms of a bizarre misconception or combination of misconceptions in predicting the behaviour of a particular program is evidencing the student behaviour which VanLehn [1983] describes in his work on repair theory. This latter rests upon the hypothesis that a student reaching an impasse in the given task, which in the work

referenced above is the subtraction process, will not usually put down pen or pencil and retire from the struggle, but will attempt to use some alternative strategy to deal with the problem. This strategy may be a similar, but slightly 'buggy' one, or one that is invented as an emergency procedure. A novice producing a prediction for the behaviour of a program and who is unsure of the control flow process can be described as facing a similar predicament when a Prolog subgoal fails. Knowing, or not knowing, as the case may be, that backtracking plays a significant part in the behaviour of a program, the student, in producing a faulty interpretation may well be attempting a 'repair' as a way of resolving the current impasse.

It can readily be seen that the search for answers to such questions is a non-trivial task. However, this does not necessarily indicate that the error analysis approach should be abandoned. Answers to some of these questions may well be found in a closer investigation of teaching strategies and tutorial actions. Meanwhile the benefits that can be derived from error analysis should be exploited in the situations where in doing so there is likely to be a reasonable gain in understanding of students' problems. In this study we have only briefly explored the possibility of multiple bugs (see chapter five for a full discussion of the errors which were modelled), the two compound errors shown below being examples of such bugs that would be relatively amenable to detection should they present themselves in the students' answers.

Redo from left preserving pointers

This error can be seen as a combination of two previously described bugs, 'redo from left' and 'one pointer per clause'. Given the program:

p if a & b & c.

a1.

a2.

b.

and the query **p**, after the failure of **c**, following the 'redo from left' model, control flow would be seen as returning to the leftmost subgoal **a**, which can be resatisfied by **a2** and then, following the 'one pointer per clause' misconception, the fresh call to prove **b** would mistakenly be predicted as failing.

try p	try a succeed	try b succeed	try c fail	try a succeed	try b fail
try b fail					

Facts first in try once and pass

Here, given the program:

p if a & b & c.

a1 if x.

b.

a2.

and the query **p**, the interpreter is correctly seen as proceeding from left to right, but incorrectly, seen to scan the clause **a2** before the clause **a1** and then on the first failure of **c**, to terminate the search. The prediction of the interpreter's steps in executing the above program would then be:

try p	try a succeed	try b succeed	try c fail	p fail
-------	------------------	------------------	---------------	--------

Error exclusion

As a result of designing the programs so that it would be possible for each of the above errors to be apparent in at least one of the problems set, particular errors were necessarily precluded from appearing in

certain problems. The 'facts before rules' error, for instance, could only appear in problem five (fig.2.18), since this problem was designed to highlight that particular misconception and was the only problem which included a rule and a fact with the same head.

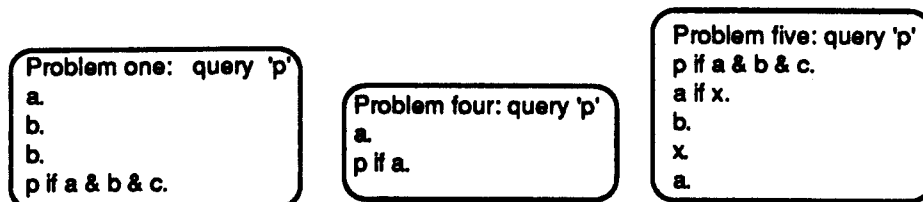


Fig.2.18. problem design

The 'one pointer per clause' misconception would not be apparent in problem one (fig.2.18), because on backtracking to 'a', an attempt to resatisfy it fails and 'b' is not called again. In problem four (fig.2.18), none of the misconceptions described earlier would be seen because backtracking does not occur.

Predictions

In carrying out the experiment, several aspects of the results were of interest. One interest lay in learning if the results would validate the hypothesis [Fung et al 1987] that Prolog's control flow is a serious source of misconceptions for the novice. Others were to discover whether there was corroboration of 'established' bugs in control flow models, if any of the bugs would be evidenced which had been informally observed but for which there is not as yet any formal empirical evidence, if certain models appeared more consistently than others.

In addition it was of interest, though not a major focus, to note if previous programming experience was relevant to Prolog control flow

errors, and/or if students who had completed all the relevant exercises in the book-based course showed any gain in understanding Prolog's control flow over those who had not. Questions relevant to this were therefore included in the questionnaire

2.2.3. Results

This section reports on the outcome of the experiment. It considers the stability of these errors across the range of problems and in the predictions of individual students. In doing so, it initially takes an overall look at the number of errors produced, their types and distribution, and relates these to earlier empirical results. It subsequently looks at the data relating to the occurrence of error types in each problem and the frequency with which each student made particular errors. It also notes the inconclusiveness of any relationship between the results of students who had completed all the course book exercises and those who had not, and indicates the pattern of results in relation to any previous programming experience which the students had acquired.

Errors in total

Within the context of the relatively small number of empirical studies of novice Prolog programmers' errors which have previously been undertaken, the predictions given by many of the students indicated the misunderstandings of the Prolog interpreter which had been expected.

Some predictions corroborated established misconceptions [Coombs and Stell 1985], [Taylor 1987], while others replicated misconceptions which

had been informally observed in novices' work, but not to date empirically supported [Fung et al 1987]. Appendix A2 contains a complete summary in table form of the type(s) of error which each student's predictions showed, of how many correct predictions each student made, of their programming experience and whether they had completed all the exercises set in the preparatory book-based introduction to Prolog. Appendix A3 contains a complete record of the erroneous predictions made by each student in each problem.

In addition to those backtracking errors described above and some errors whose patterns were able to be interpreted with reasonable certainty, there were many other errors which at this stage were not able to be clearly identified. Although some of these presented patterns suggestive of a particular, if incorrect, model of Prolog control flow, they need further investigation before a meaningful interpretation can be attempted. In the following discussion this type of error has been classified as 'unidentified' and unless relevant to the point under consideration is not discussed here in any detail.

The bar chart below (figure 2.19) shows the total number of errors which occurred in the predictions of control flow in the six problems set. Out of this total, 47% were classed as unidentified and 53% as identified.

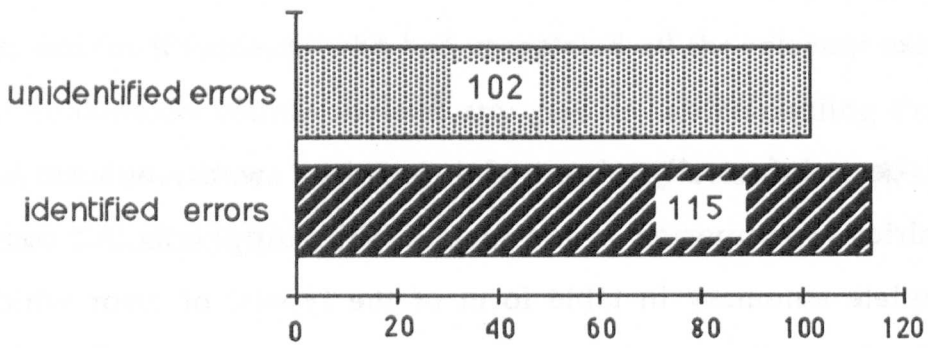


Fig.2.19. Breakdown of total number of errors (217).

The following graph (fig.2.20) shows the number of errors which each of the thirty-two students showed in their predictions.

The largest number of errors made by any one student was sixteen and only one student made no false predictions whatsoever. The median number of errors per student was seven, the mean was 6.8.

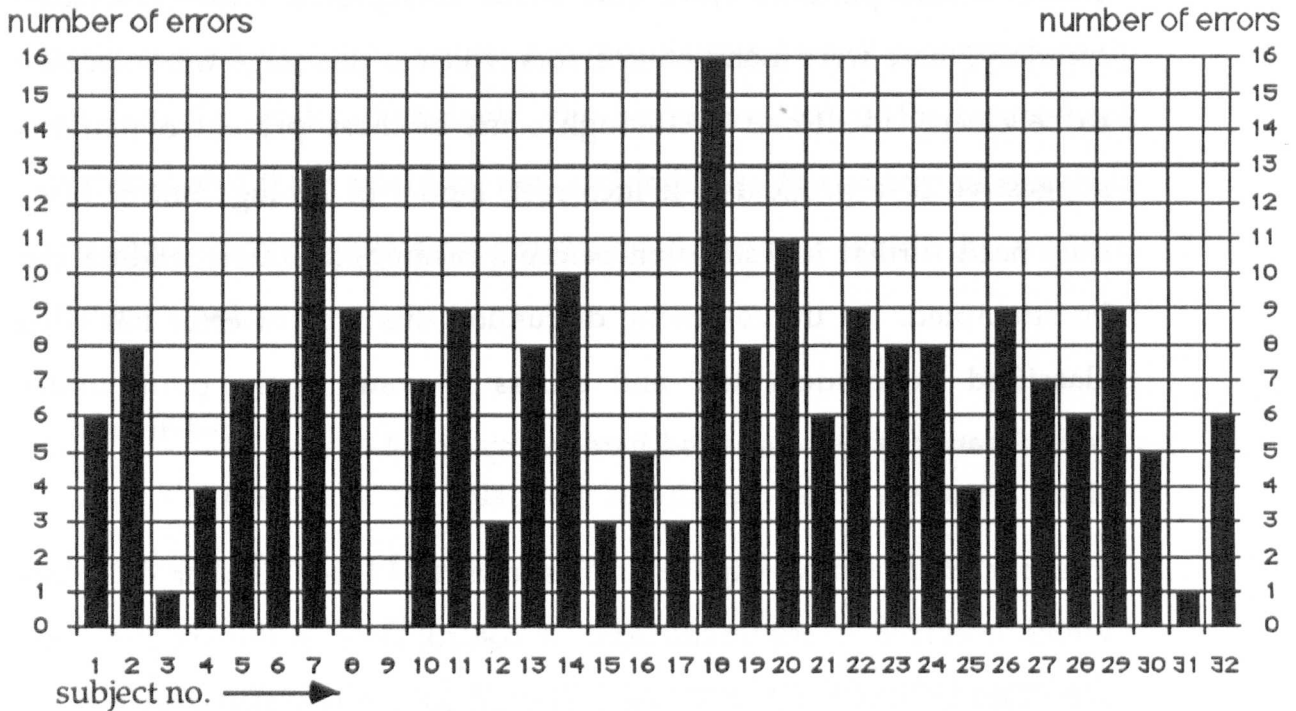


Fig.2.20. Number of errors made by each student

The chart on the following page (fig.2.21), shows the spread of numbers of errors which students made. While a few students have made a low number of false predictions and equally, a few have made a relatively

high number of false predictions, most students have made between six to nine false predictions.

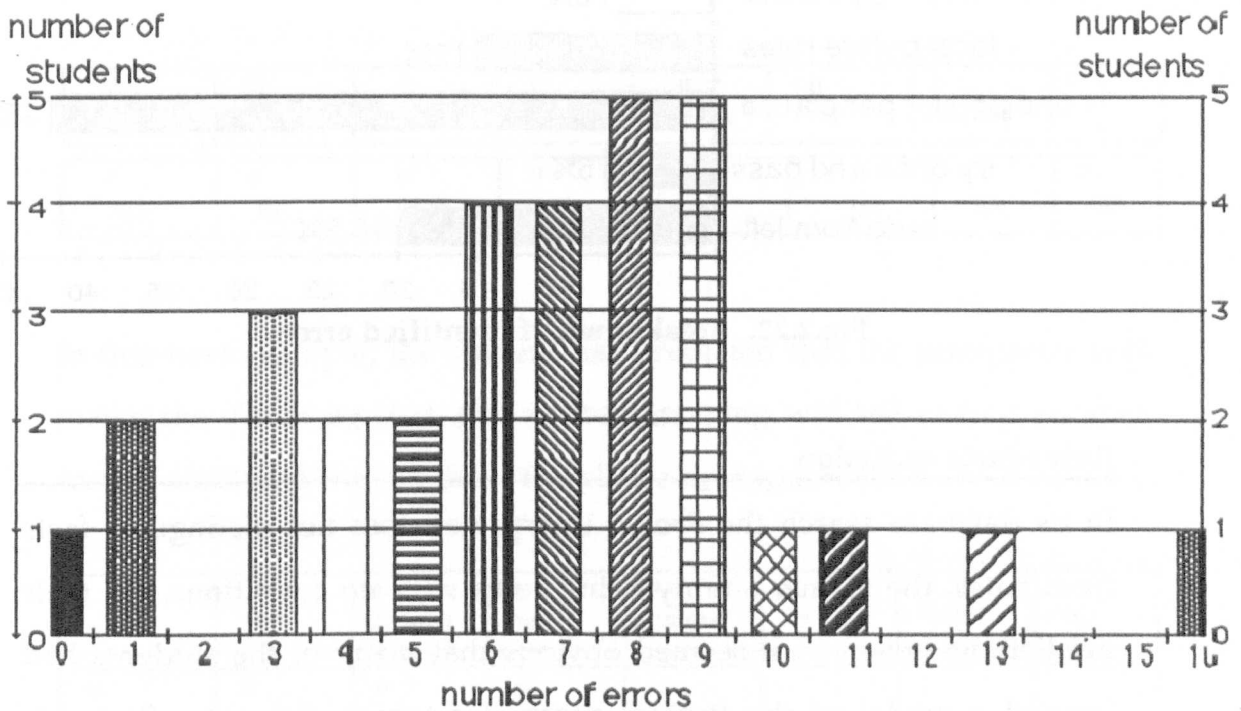


Fig.2.21. Distribution of numbers of errors made by students

Types of error

The term 'identified errors' is used to denote those errors which follow a pattern consistent with the five control flow errors described in the first part of this paper, plus two additional errors which appeared with a certain consistency in the students' predictions. These two errors are discussed in the paragraphs following fig.2.22 overleaf, in which they are referred to as 'rules-facts exclusion' and 'meta-knowledge'.

Out of a total of 115 identified errors, the following bar chart shows how many times each type of error which was found in the students' predictions. The percentage number marked beside each error type bar indicates its share of the overall total of identified errors.

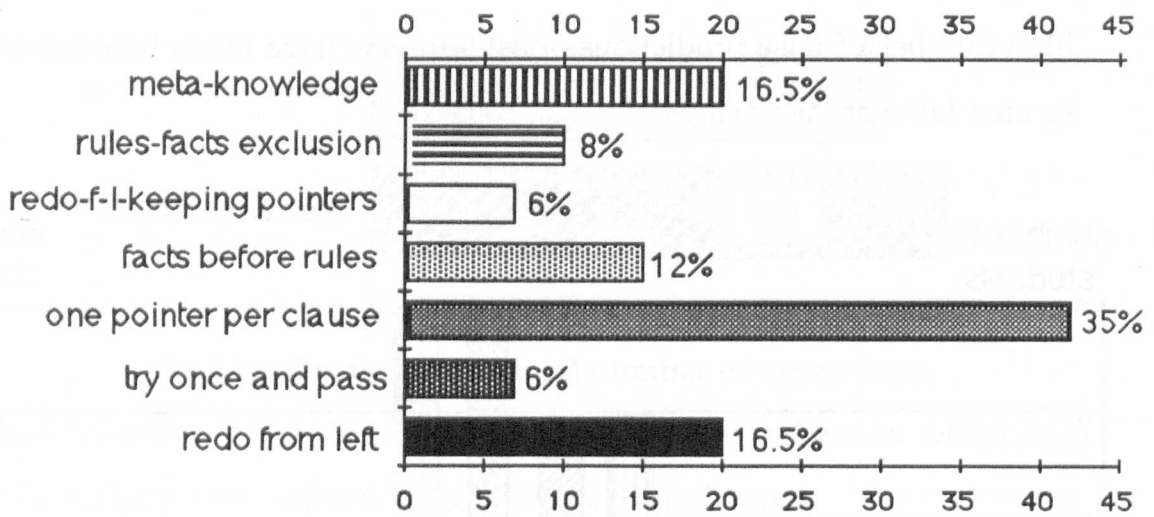


Fig.2.22. Breakdown of identified errors

Rules-facts exclusion

In its database search the Prolog interpreter does not distinguish facts from rules, the former simply being rules with no conditions. In their predictions however, it seemed obvious that many of the students had formed a model of the Prolog interpreter which did not reflect this correctly. Problem five was designed to test whether students mistakenly presumed that the interpreter checked facts before rules. In predicting the goal search in this problem, many of the students seemed to exercise a rules-facts exclusion. Overleaf, to illustrate this, are predictions given by two of the students evidencing this misconception. In this problem the students were asked to predict the steps the interpreter would take in answering the goal query p , given the following program:

```
p if a & b & c.
a1 if x.
b.
x.
a2.
```

In this example the student has predicted that clause **a₁** is tried first (correctly), but has predicted that on backtracking the interpreter will fail to find another way of satisfying **a**. The clause **a₂** is ignored.

p try	a try	x try	x succeed	a succeed	b try	b succeed	c try
c fail	b try	b fail	a try	x try	x fail	a fail	p fail

In this next example, the student has predicted that the interpreter will match the clause **a₂** first, and on backtracking will fail to find another way of satisfying the call to **a**. The clause **a₁** is ignored.

p try	a try	a succeed	b try	b succeed	c try	c fail	b try
b fail	a try	a fail	p fail				

It seems that a model had been formed in which the interpreter makes a distinction between rules and facts, though this distinction is not necessarily determined by database ordering. This model predicts that if an attempt to match a subgoal succeeds using a fact, then subsequent attempts at matching will not make use of a rule and vice-versa. Ten students out of thirty-two exhibited this model of the interpreter. It seems that the interpreter is seen as having a capability of matching rules or facts, but not both in the course of any one particular goal search. This adds strength to the hypothesis that a model has been formed in which the interpreter distinguishes between rules and facts.

Meta-knowledge

In all the problems except number four, the eventual outcome of the program would be a failure to prove the given query, the point of interest lying in the prediction of the search made by the interpreter in order to reach that conclusion. It was interesting to find that in several cases, although the students concerned predicted the correct conclusion, it was apparent that they did not have an accurate model of how the interpreter reached this conclusion. Given that with one exception the programs would ultimately fail, a number of students predicted this failure at a point where the interpreter was still able to resatisfy subgoals. This would imply that in prematurely failing such programs they had used a level of reasoning based on their knowledge of the real world rather than following through the backtracking process of the interpreter. To them it was visually apparent perhaps that at a certain stage there would be no further point in backtracking, since the program would necessarily fail if a certain fact could not be proved. While for a human being this is a reasonable way of analysing a problem, the interpreter does not possess this meta-knowledge. As pointed out in Fung et al [1987] in the description of this error, which is termed 'meta-analysis' and also noted by Taylor [1987], there may well be some relation between this phenomenon and the 'superbug' phenomenon discussed by Pea [1986]. Pea points out that in computing, the novice programmer has no analogy to draw upon except that of issuing instructions which are to be acted upon by another human. Since in such a person to person situation, one takes for granted that the other person has a store of implicit knowledge to bring to bear upon the subject, the novice could be unwittingly attributing to the interpreter implicit knowledge of a clever search strategy that in reality it does not have. The term 'meta-knowledge' used here indicates that a

student has exhibited the symptoms of this type of error. Below is a typical example of a prediction of the interpreter's actions by a subject who may well be suffering from this misconception. Given the following program:

a.
a.
b.
p if a & b & c.

and the query to prove p

p try	a try	a succeed	b try	b succeed	c try	c fail	b try
b fail	a try	a succeed	p fail				

Note that although the second call to a succeeds, the student has prematurely failed the program at that point without even attempting to retry b. The normal action of the interpreter of course would be to proceed to try b again and then c and once more backtrack unsuccessfully to p before finally failing.

Distribution of error types

In this section we look at the distribution of identifiable errors, including the two described above, which students showed in their answers.

Corroboration of established misconceptions

Both the 'redo from left' and 'try once and pass' misconceptions which have been identified in previous studies, [Coombs& Stell 1985], [Taylor 1987] were found. Interestingly, the small percentage of 'try once and pass' errors seems to indicate that even in cases where students were not sure of the interpreter's actions, the majority were aware that some

backtracking process would take place. Comparatively few showed evidence of a control flow model which omitted the process entirely. This is of particular interest in view of the results of a second empirical study which was undertaken in the following year and which had strongly contrasting results in this respect (see chapter six).

Evidence of other expected misconceptions

The 'one pointer per clause' misconception, observed informally but not previously empirically supported was present in a comparatively large number of predictions. A factor contributing to this large number was the consistency with which some students predicted this behaviour of the Prolog interpreter in each of the problems in which this misconception could feasibly be applied. Problems one and four excluded the possibility of displaying this model of the interpreter, so are not shown in the figure below (fig.2.23).

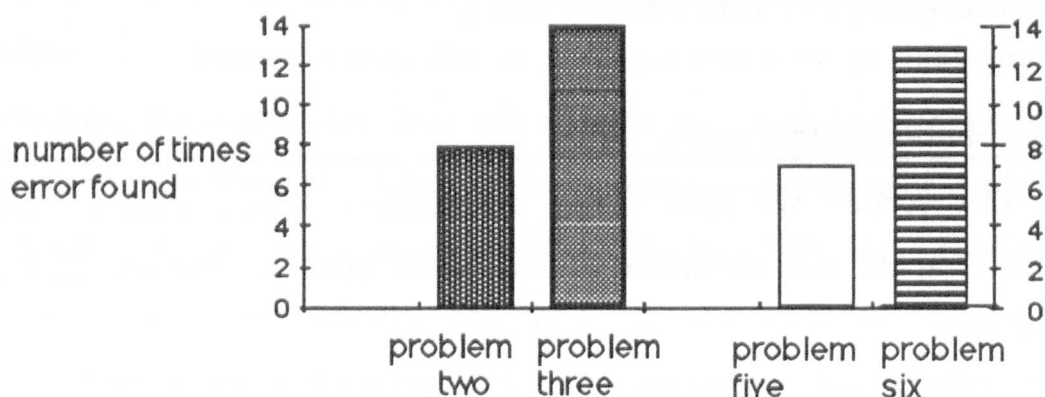


Fig.2.23. distribution of 'one pointer per clause' errors

A model of the Prolog interpreter which indicated the misconception of 'facts before rules' also seemed to be supported by the results, since fifteen out of thirty-two students predicted that the interpreter would choose to match a fact before a rule with the same head, even though in the example program the rule appeared before the fact.

The seven instances in which the multiple bug 'redo from left preserving pointers' occurred (see fig.2.22) were accounted for by the predictions of three students. One student showed this error in four problems, another in two problems and the third displayed it in one problem only.

Evidence of unexpected misconceptions

The two bugs which we felt merited classifying as 'rules-facts exclusion' and 'meta-knowledge' and described above, appeared with some consistency, ten occurrences of the former and twenty of the latter being noted. 'Rules-facts exclusion' necessarily appeared solely in problem five, the only problem to involve rules and facts, while, as can be seen below, examples of 'meta-knowledge' misconceptions appear in all except problem four, the problem which involved no backtracking.

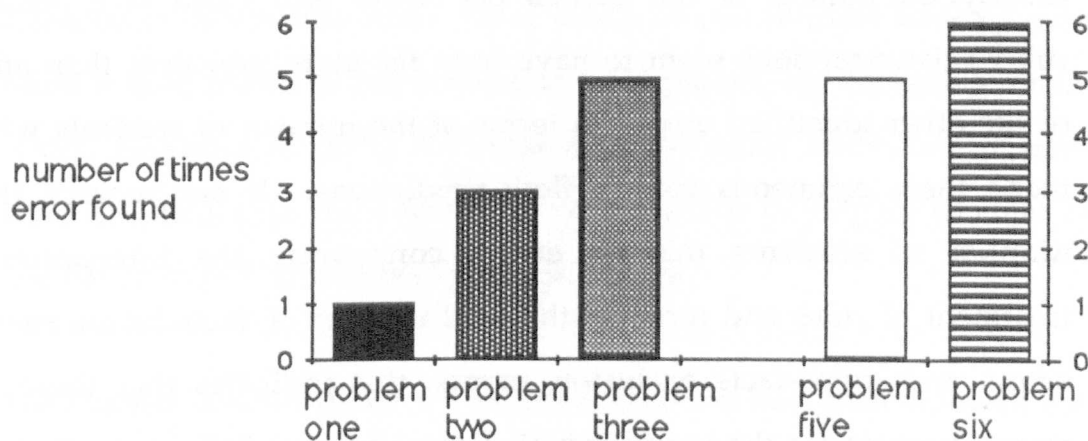


Fig.2.24. distribution of the 'meta-knowledge' misconception

Distribution of error types among subjects

As opposed to the number of errors which each student made, the bar chart below (fig.2.25) shows the number of students who made each of

the types of errors mentioned, e.g. 6 students showed the 'redo from left' misconception.

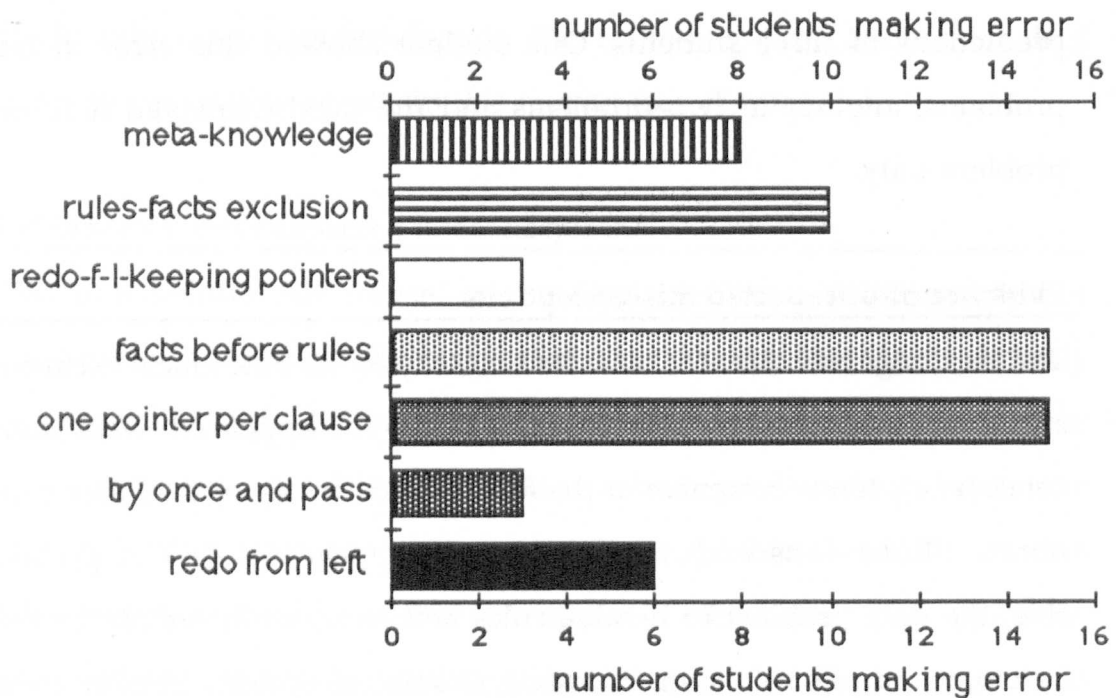


Fig2.25. number of students related to error type

Clearly, the number of 'one pointer per clause' errors and 'facts before rules' misconceptions seem to have been far more prevalent than any of the other identified errors, in terms of the number of students who made these erroneous control flow predictions. If one regards the number of students making errors concerning the interpreter's treatment of rules and facts i.e. the total number of 'facts before rules' errors and 'rules-facts exclusion' errors, this indicates that there is strong support for the hypothesis that some novices believe the Prolog interpreter distinguishes between facts and rules in its database search. Similarly, looking at the number of students making the 'one pointer per clause' error, there seems to be evidence that this aspect of control flow presents difficulties for them.

Bug stability

There are two aspects of the distribution of bugs which are of interest in this respect. One is the distribution of error types across the six problems. The other is the stability of a bug type across each student's predictions. The term 'stability' is used here to refer to how consistently a student used a particular buggy model of the interpreter in each of the problems in the process of completing the questionnaire. We look first at the distribution of error types over the range of problems.

The following matrix shows how many students made which of the expected errors in each problem. When designing the experiment each problem was chosen with a potential misconception of the interpreter in mind. The purpose of this was to allow an opportunity for that particular error to occur, since as mentioned above, certain programs could exclude certain errors. This is analogous to making sure that in a selection of subtraction sums there would be one or more which would allow a false model of the process to be apparent, e.g. including a sum which necessitated using decomposition or equal addition. If none were included, then there would be no possibility of judging whether or not the student had mastered those processes.

The shaded boxes indicate which misconception had been 'allowed for' in each problem, e.g. in problem three it was ensured that the program involved a fresh call to a previously failed subgoal, so that the 'one pointer per clause' misconception could occur, in problem five the program included a fact and a rule with the same head in order to test the hypothesis that some students believe the interpreter scans facts before rules.

number of students displaying particular error (s)					
problem number	Redo from left	Try once and pass	One pointer per clause	Facts before rules	Redo f-l preserving pointer
1	1	1	0	0	0
2	5	2	8	0	1
3	4	2	14	0	2
5	3	1	7	15	3
6	4	1	13	0	1

Fig.2.26. problems and expected errors

This could raise the question of whether this would encourage the appearance of a particular error in certain problems, or constrain the subjects to making that error in one particular problem only. However, this proved not to be the case, with one obvious exception i.e. erroneous predictions concerning rules and facts. Since only problem five contained a rule, in this case, the 'expected' error could only be found in that slot. The table above shows that, with this exception, each error was, in general, fairly well distributed over the problems in which it could occur. In each shaded box, the total number of students displaying the 'expected' error in that problem is not notably higher than in other problems. Problem four was not included in this table, since it was not expected that any of these errors would occur in it and it did not, in the event, account for any of the identified errors. The bar chart overleaf (fig.2.27), makes the same point more graphically. Bearing in mind the exception of problem five, mentioned above, problem selection exercised no significant constraint upon the distribution of errors across the problems.

'Redo from left' for example, was found in four subjects' answers to problem one, in five subjects' answers to problem two, in four subjects'

answers to problem three, three subjects' answers to problem five and four subjects' answers to problem six.

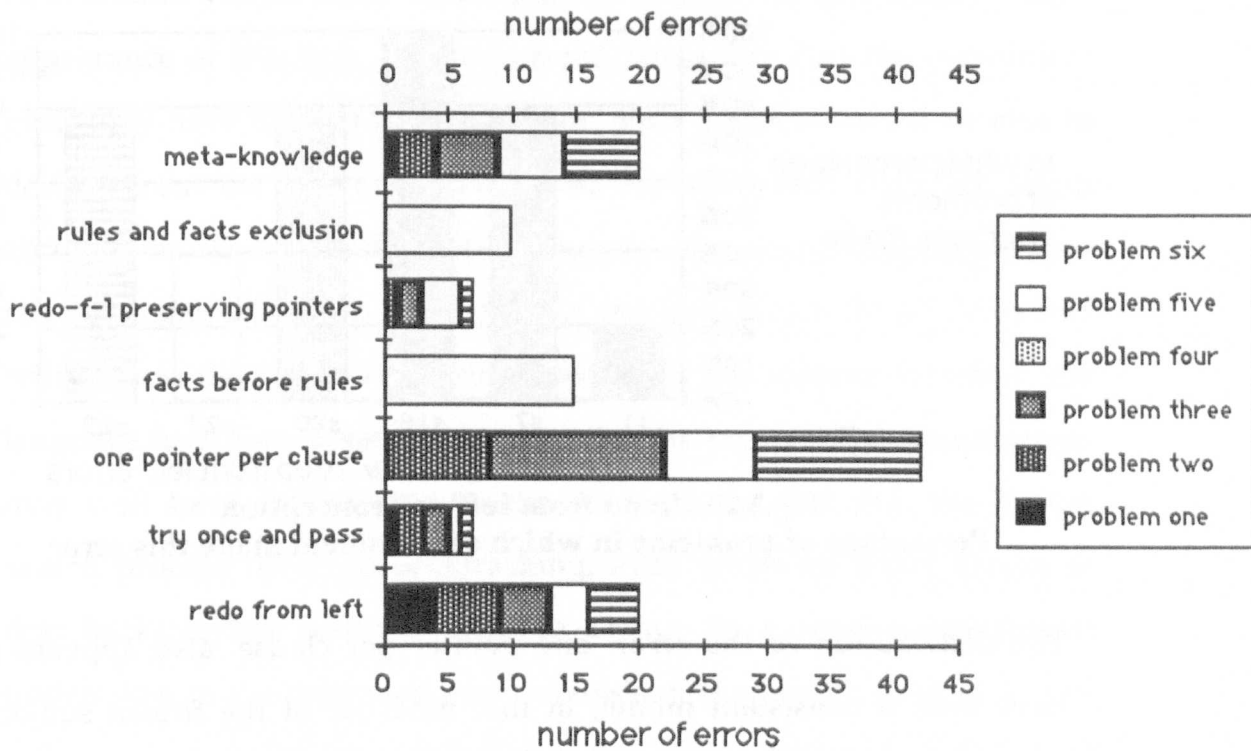


Fig.2.27. distribution of errors across problems

This leads on to the second point regarding the stability of bug distribution, that of stability in the individual student's predictions. Tables showing the percentage of occurrence of each of the individual error types in the students' predictions across the range of problems are included in Appendix A4. Two of these are shown below. They indicate the percentage of problems in which each student displayed that particular error.

The 'redo from left' model of the interpreter was consistent among those students who displayed this error. Among the six students concerned, four showed the error in at least four out of the five problems in which it could have been apparent. Of these six students, three also showed the 'one pointer per clause' error, accounting for the

occurrences of the multiple bug 'redo from left preserving pointers' described earlier.

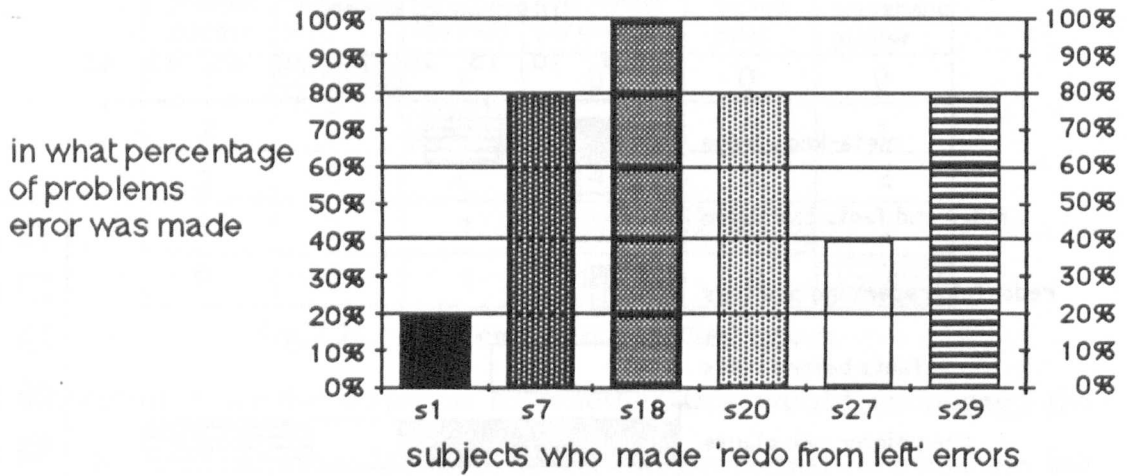


Fig.2.28. 'redo from left' misconception
Percentage of problems in which each student made this error

The distribution of the error 'one pointer per clause' also appears to have been a consistent model, in that nine out of the fifteen subjects concerned made this error in at least three out of the four problems in which it could have appeared.

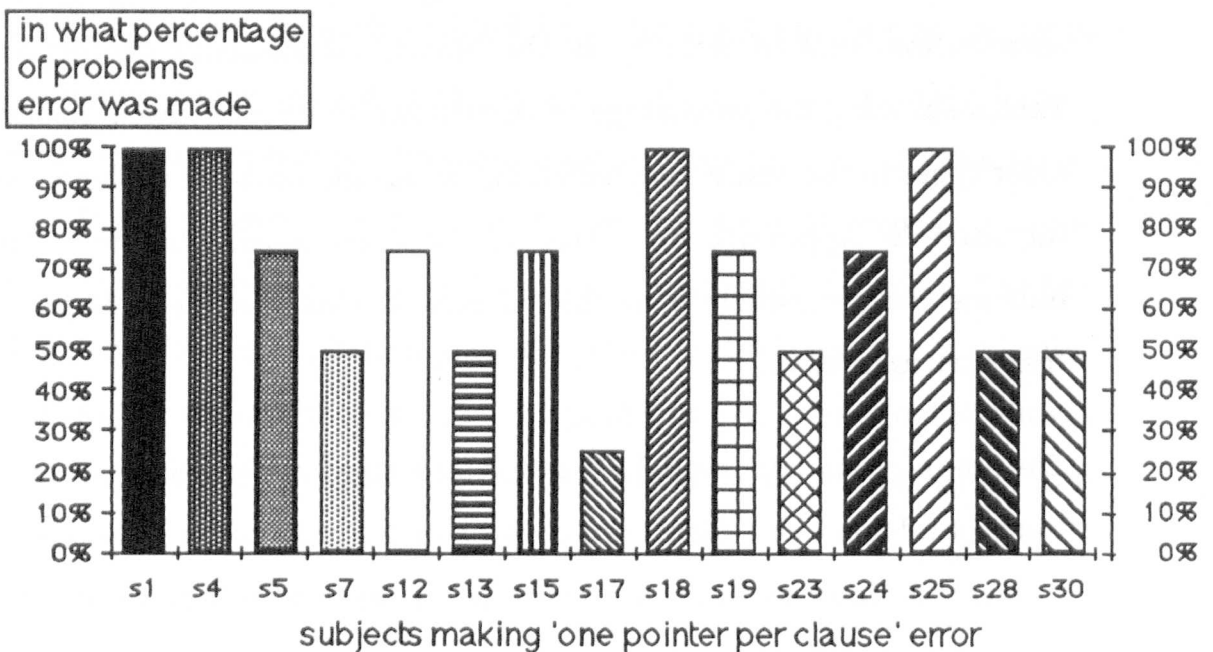


Fig.2.29. 'one pointer per clause' misconception
Percentage of problems in which students made errors

As noted in an earlier section, relatively few students showed the 'try once and pass' misconception noted by [Coombs & Stell 1985]. There are two factors which may have played a part in the relative non-appearance of this bug. There is some possibility that the experiment itself may have discouraged this error, since subjects would be able to deduce from the examples given that the interpreter does not finish abruptly on the first failure of a subgoal. The other factor is perhaps related to a combination of the type of program which the students had written and debugged for their projects and the manner in which the language had been presented to them by the tutors. This combination may well have emphasized to the students the fact that the Prolog search process involved backtracking, even when the exact nature of that backtracking was not clear to them. In a similar experiment undertaken in the following year, analysis of the results showed a very different pattern for the occurrence of this error. This will be discussed more fully in chapter six when those results are reported

Previous course experience

As noted in section [2.2.2] one of the secondary intentions in the experiment was to note if the group of students (group A), who had completed the exercises set in the book-based introductory course for Prolog would make less errors than those who had not (group B). This hypothesis was based on the possibility that completion of the exercises provided in the introductory course book, would, by reinforcing the procedural aspect of Prolog's database search, help to reduce the number of errors made. The difference however, between the results of these two groups was not significant (Mann Whitney test) and did not allow an unambiguous interpretation of the data.

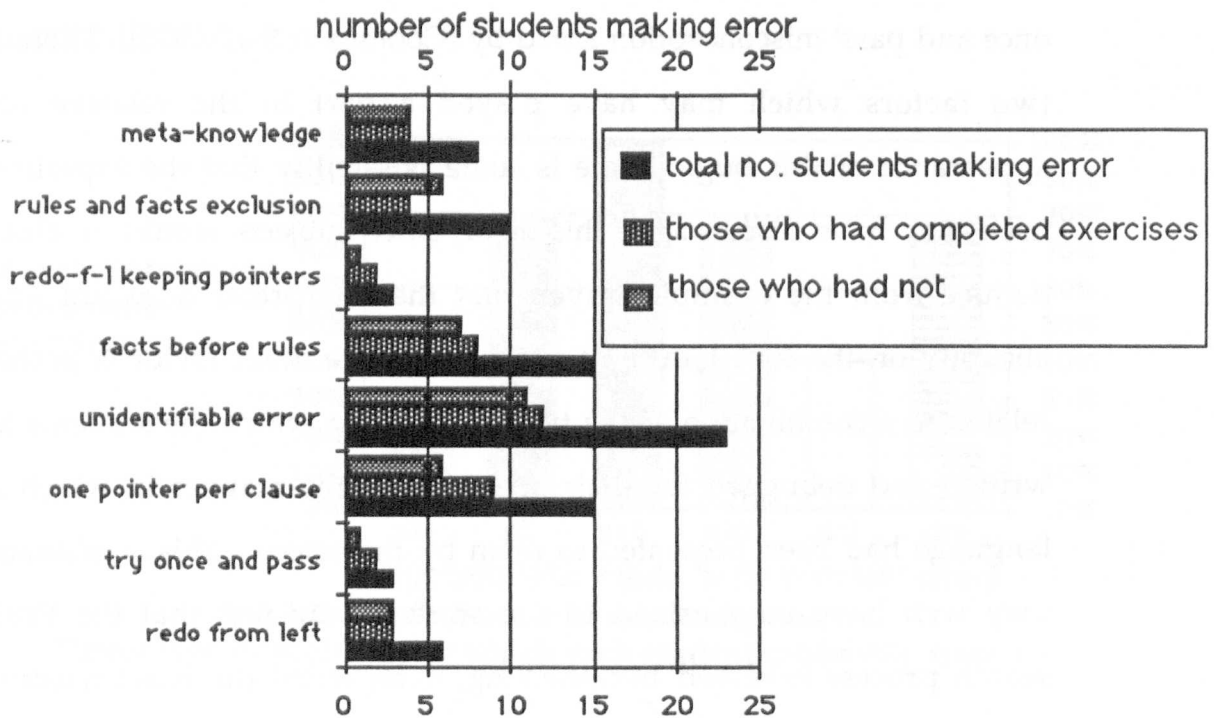


Fig.2.30. Comparison of errors
(made by students who had completed book-based exercises prior to summer school and those who had not).

There were, however, interesting (but statistically non-significant) differences between the two groups. Out of a total of seven identified misconceptions and the category of unidentified errors, equal numbers of students in each group evidenced 'meta-knowledge' and 'redo from left errors'. In all the other categories of errors, with the exception of 'rules-facts exclusion' (where rather less group A students made errors, as had been hypothesized), the number of group A students showing each type of error was slightly greater than the number of group B students.

Since no investigation was made into why the book-based exercises had not been completed, the reason for this result must remain unclear. One hypothesis is that the students who felt they had grasped the

essential points of the course did not bother to complete the exercises. For the time being however this must remain an untested hypothesis.

Previous programming experience

Number of students showing evidence of the errors noted and relation to programming experience					
type of error	total no. students showing error	students with no prog.exp n = 10	students with Prolog exp. only n = 9	students with prog. exp. but not of Prolog n = 9	students with exp. of Prolog and other language(s) n=4
Redo from left	6	2	2	2	0
Try once and pass	3	1	1	1	0
One pointer per clause	15	5	6	3	1
Unidentifiable error	23	7	8	6	2
Facts before rules	15	3	4	5	3
Redo f-1 plus pointers	3	1	1	1	0
Rules and facts excl.	10	1	3	5	1
Meta-knowledge	8	3	3	2	0

Fig.2.31. Errors related to programming experience

The number of students who had programming experience of both Prolog and another language was too small to allow any statistical analysis of the data, so it is not possible to point to any significant differences between these subjects and those with less experience.

There is, however, an interesting trend in the results of the four groups. It is remarkable that the students with previous experience of Prolog, or

previous experience of one language only, made more errors than those who came to the course with no programming experience whatever.

This can be seen more clearly in terms of the average number of errors which each group made, shown below in fig.2.32. Unfortunately there was no way of investigating this phenomenon within the limits of the study, since there are several factors which could have contributed to this result but which could not have been taken into account. Apart from factors such as individual differences in ability, it is also possible that the way in which the previous programming experience was gained, for example, could have had an effect upon the performance of these students. In contrast to this, the average number of errors made by those students who had used both Prolog and another language(s) prior to the course was lower than the error average of any of the other three groups. Although the small number of subjects in this group precludes statistical testing of this difference, it is a result which would intuitively be expected.

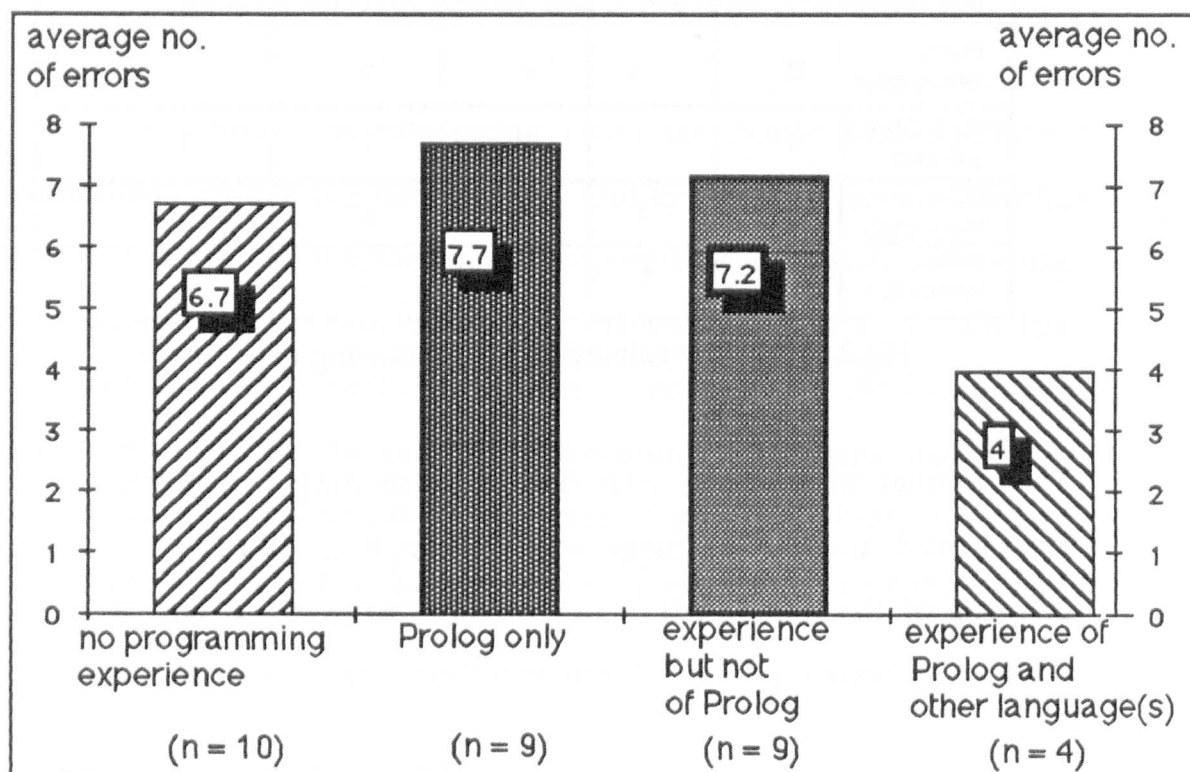


Fig.2.32. Comparison of error averages

2.2.4. Comments

As [Sheil 1981] points out, it would be unwise to make generalisations from the results of one study and apply these over the whole spectrum of novices being investigated. Different methods of teaching Prolog must certainly have some part to play in determining the kind of mental models students form of the interpreter. The results corroborate earlier work on novices' backtracking errors by Coombs and Stell [1985] and by Taylor [1987] i.e. the misconceptions 'redo from left' and 'try once and pass'. They provide data which support the existence of possible misconceptions put forward in Fung et al [1987], i.e. 'one pointer per clause', 'facts before rules', while raising questions which merit further investigation regarding the misconceptions students have of the Prolog interpreter's treatment of rules and facts, i.e. the 'rules-facts exclusion' noted in the results. It is possible that these misconceptions are a product of the way in which Prolog is often taught. As was discussed earlier, the vocabulary of teaching Prolog often reinforces, for instance, the idea that facts and rules are different. This is valid from the programmer's point of view, in that for various reasons they often merit different ordering in the database. It may be however that this emphasis on 'facts' and 'rules' over-rides the idea that to the Prolog interpreter they are both simply logical statements, the latter with conditions and the former without. A change in the description language used to explain Prolog structures to beginners could well go some way towards eliminating this particular source of confusion.

The misconception which appeared in the students' predictions and which is described here as 'meta-knowledge' is also one that needs further attention. Similar to a misconception described by Rajan [1986]

as 'real-world knowledge', it may well be, as discussed, a manifestation of one of Pea's [1986] 'superbugs'. This could prove a difficult error to tackle, being as Taylor [1987] points out:

of diverse origins and a function of assumptions on the part of the learner, teaching methods and the way in which the programming language presents itself

It is clear however that such a misconception needs to be tackled. Its manifestation in predicting the execution of a short simple program may make little difference to the outcome, and indeed can be of use when mentally simulating execution for debugging purposes. In terms of understanding larger or even slightly more complex programs, harbouring such a misconception can cause serious confusion. In such circumstances a student may well presume that at a particular juncture a goal will fail because she or he can see that a certain fact is not present in the database. The interpreter, however, continues to backtrack in a completely mechanistic fashion and may succeed in resatisfying a goal or subgoal through another search path, producing an unexpected or wrong solution to the given query.

At a level more specific to the results obtained in this study, there did appear to be a certain stability in many of the faulty models exhibited. This stability relates both to the type of errors seen across the range of students and also within the predictions of individual students. The subjects concerned were very much in the early stages of learning to program in Prolog and the results help to underline just how quickly students do form some kind of execution model. The results also indicate how common it is for students beginning to use Prolog to

develop a faulty or incomplete model of the most basic operations of the interpreter.

As an exercise in gaining insight into novice Prolog programmers' conceptions of program execution this study has provided evidence of some of the models which beginners form. While some errors which came to light raised questions for which there is no immediate answer, other errors were sufficiently well defined and their occurrence frequent enough to be considered as likely symptoms of novices' misconceptions of the Prolog interpreter.

2.3. Summary

The empirical study reported in this chapter confirmed that many novices have difficulty in understanding the highly procedural nature of the Prolog interpreter. Of the subjects taking part, the majority showed by their predictions of Prolog execution that they had incomplete or incorrect models of the backtracking process.

A selection of those models appeared with a consistency that suggested they could play a useful role in the task of developing a system of on-line analysis of novices' misconceptions of the Prolog interpreter. They provide a suitable starting point to explore the use of formal semantics in modelling control flow errors, the approach put forward in this thesis as a solution to the problem of student modelling in a diagnostic tutoring module.

The first step in this exploration is to model these misconceptions, generating them from the formal descriptions of a selection of

programs used in tutoring. These computational models would then be available for use in automatic analysis of control flow errors.as a basis for providing appropriate tutoring help to those students. Longer term they could serve as heuristics to reduce the search space in the task of on-line reconstruction of the student's perception of Prolog control flow in cases where this does not conform to one of the more common misconceptions. In the next chapter, we discuss the formalism which we have chosen for the task of constructing computational models of the errors reported here.

Chapter Three

3. Formal Models

In the context of considering the help available to novice programmers we looked briefly in chapter one at a selection of tutoring systems for programming languages. We refer to them again in this chapter in order to show the need to explore another approach to student modelling and put forward our reasons for basing this approach on a formal semantics. The focus of this chapter is a discussion of the formalism which we have chosen as a basis for that exploration. In it we outline the ideas central to that formalism and discuss the features which make it attractive for the task of formalising novices' misconceptions in the domain of programming language.

3.1. Modelling students

As we pointed out earlier, it would seem obvious to employ the computer itself to assist in the teaching of programming languages since the student, by virtue of the domain, is in a position to take direct advantage of any on-line help [Du Boulay & Sothcott 1987]. The research problems faced in providing such help, are, as we have seen, challenging. In his 'framework for adaptive teaching' [Hartley 1973] stipulated that computer-assisted teaching systems must incorporate four components. These were knowledge of the domain, knowledge of

the student, a set of teaching operations (i.e. a curriculum and a plan for implementing this curriculum, be it through a directive approach or one based on enquiry) and lastly, a set of teaching strategies, the use of which should be directed by the response of the student. Of these, the task of representing the knowledge of the student, i.e. student modelling, has, in particular, proved to be a difficult one. In forming a model of a student's knowledge state, error diagnosis is only one of many indicators which a human tutor can use to assess learning. In computer tutoring the means of assessing a student's progress are necessarily more limited, since assessment can only be based on the information received and inferred from the user's input. Although its use, as we have previously discussed, is not without limitations, error diagnosis remains a significant method of on-line assessment of progress and plays an important part in the student modelling techniques of any system for tutoring programming languages. Careful error diagnosis not only allows the more superficial syntactic mistakes to be signalled, but can be used as a means of providing insight into underlying misconceptions held by a student concerning the language being tutored. In the following section we look again at two of the systems which we discussed earlier and consider how they have approached this difficulty of interpreting learners' input .

The approach adopted in the system 'Greaterp' [Anderson & Reiser 1985] is a pragmatic one. Using prestored programs and their prestored 'correct' solutions, this system constrains the user to following a particular learning path. Should the student at any stage input code which does not match the prestored model solutions, an error is signalled. As we pointed out in chapter one, no differentiation is made between code which is syntactically faulty and code which, while

syntactically correct, does not match the prestored solutions. In both cases the code is treated as incorrect. This close constraint upon the student to follow a particular learning path poses problems for the construction of a deeper level student model related to the individual's understanding or misunderstandings of programming concepts and program execution. The system contains only prestored information about expected correct and incorrect code and contains no representation of 'knowledge' about program code. Consequently it cannot offer any explanation other than prestored text. If this 'knowledge' of the domain were able to be incorporated, it could then be used to give more directly relevant tutoring information concerning the error or the misconception which underlay the error.

The tutoring system for Pascal, 'Proust' [Johnson and Soloway 1985], which we looked at in chapter one, attempts to analyse the student's intentions underlying the input code (in this respect 'intention' can be interpreted as 'the programming goal'). The system suffers however, from handicaps in its approach to student modelling. On the one hand, as with 'Greaterp' a great deal of information for each programming problem must be pre-stored. On the other hand and probably more important, is that the system itself, as in the 'Greaterp' system, has no representation of 'knowledge' of the language being taught. This 'knowledge' is an essential component of a tutoring system if it is to have explanatory power. Ideally a system should be able to trace the correct behaviour of the constructs of the language being taught, in order to contribute to the goal of analysing input meaningfully. One way of achieving this is to incorporate in the system an exact and unequivocal interpretation of the possible behaviours of the programming language being taught.

A step in this direction has been proposed by [Reiser, Friedmann, Kimberg & Ranney 1988], in order to remedy the implicit lack of system knowledge discussed above with regard to the Lisp tutor 'Greaterp'. It is suggested that such a system be augmented with knowledge of the intermediate programming processes involved in the programming tasks. To do so they propose the addition of a problem solving component which gives a more finely grained analysis of the execution process of each programming task, which could then be used for reasoning in relation to the student's input. While this is an approach to the goal of incorporating knowledge of the domain into the system, it does not address the problem fully. In a sense this approach tackles the problem at one remove. It attempts in its rule-base to give a more accurate description of the actual execution of a programming goal, in order to afford scope for reasoning about the shortcomings of student input or the need for a certain approach to a programming task. As described in their paper [Reiser et al 1988] this relies on a quasi natural-language description, in terms of input, output and goals, of the process concerned. While this is certainly a direction which can usefully be taken to provide tutoring systems with the level of explanatory power necessary, long term it would seem more appropriate to tackle this description at the level of the language itself. A precise semantic description of the language which incorporates the execution model of the language would overcome the problem of scope of explanation. At present this is limited by the strategy of describing each process individually as it occurs [Reiser et al 1988], whereas if it were described at the more global level of the language the scope of reasoning about the programming code could be correspondingly extended. Additionally, such a description demands an accuracy and correctness

which is best supplied by a formal semantics. While there have been instances where programming languages have been defined in English, this can result in ambiguity and inconsistency, as has been pointed out by [Knuth 1967] in referring to ALGOL 60. To avoid this demands the mathematical rigour of a formal description. The underlying mathematics of a formal semantics, to quote Hoare [1985], provides

a secure, unambiguous, precise and stable specification of the language to serve as an agreed interface between its users and its implementors.

In the next section we look briefly at the role of formal descriptions in relation to programming languages.

3.2. Formal descriptions

While Backus-Naur form has provided an adequate way of formally describing the syntax of programming languages since at least the late 1950's [Wirth 1963], formal semantic descriptions of programming languages have taken longer to develop. Nevertheless they are equally essential, not only as discussed here in terms of use as a tool for precisely describing the execution and correctness of programs in tutoring systems, but in a broader context of reaching a viable comparison for standardization purposes between the different implementations of a programming language or between the properties of different programming languages. In such contexts semantics

are as essential to the objective of language standardization as measurement and counting are to the standardization of nuts and bolts. [Hoare 1985]

There are also several different kinds of formal semantics which have been developed, the most well-known of these being 'denotational' 'operational' and 'axiomatic' semantics. Each has grown out of a concern to achieve a particular goal in the semantic description of programming languages. Put very simplistically, denotational semantics are concerned with defining languages, axiomatic semantics with developing rules of inference for reasoning about programs and operational semantics with describing the implementation of languages. A succinct appraisal of the potential of each of these different kinds of semantics as a basis for a formalisation to be used in programming language tutoring systems can be found in [Elsom-Cook 1984], in which the author quite rightly concludes that each of the above-mentioned semantics, taken alone, has drawbacks in the context of teaching a programming language. This arises because any formalism to be used in a tutoring system for programming languages must serve to achieve multiple goals, as we discuss in the following paragraph.

In the formalisation of Prolog errors which is currently being investigated, the formal semantics chosen must serve to define the language, to test the correctness of programs and to describe its implementation. For the purposes of a tutoring system such a semantics must possess not only the ability to generate a description of the 'correct' model of the language concerned but also the ability to generate a description of incorrect models. It must also have the capability of affording a semantic explanation of the student's input. To do this requires a notation which supports an intuitive interpretation of program execution. This being so, we are looking for a formalism possessing the properties necessary to achieve each of these goals whilst

preserving the mathematical foundation which distinguishes a formal semantic description from one that relies solely on intuitive description. In recent years the potential power of parallel programming, with the attendant questions it raises in relation to running concurrent processes, has stimulated interest in, and development of, formalisms capable of fulfilling such aims. The results of this interest also has important implications for sequential languages. Two significant advances in this area have emerged in the work of Milner [1980] and Hoare [1985]. The formalisms developed in these works both have a mathematical basis of a systematic collection of algebraic laws and are suitable vehicles for reasoning about program design, specification and implementation. In addition to this mathematical basis both support an intuitive interpretation, a significant advantage in relation to its use in a tutoring system. However it is the first of these formalisms, Milner's Calculus of Communicating Systems (CCS) [Milner 1980], which is of major interest to us here, since it specifically has the potential of formulating different levels of equivalence of processes or programs based on observations that can be made about their behaviour. Within the context of interpreting student input this is an aspect of particular interest. The key ideas of his work are described in the following section.

3.3. A Calculus of Communicating Systems

We give here a broad view of the ideas and principles originally developed by Milner. For a fuller and more technical description, the reader is referred to [Milner 1980]. His purpose in this work was to develop a calculus which would offer a way of describing concurrent systems with accuracy and would provide a means of reasoning about

those systems. CCS serves as a framework for building different models and undertaking a comparison of these models at differing levels of abstraction. Among the applications he envisaged was the semantic description of parallel programming languages, the work of Beckman [1987] on parallel Prologs being an interesting example of this area of application. Underlying the calculus is the concept that the behaviour of a system can be determined by observation. The two key ideas central to this are (i) synchronised communication between the agents or machines of a system and (ii) the notion of observational equivalence of behaviours of systems. In the following sections we look briefly at each of these ideas.

3.3.1. Synchronised communication

The first idea central to the calculus is that a system in CCS is seen as a set of linked communicating agents or machines (the two terms are used synonymously by Milner). Each agent, or machine, of the system has certain actions which it may perform. The description of the actions of each of the agents in a system forms the basis of the description of the whole system. Imagine for instance, a miniature retail system with only two components, Fred the customer and Hatter the shopkeeper. A simplified approximation of the individual set of possible actions of agent Fred and of agent Hatter, the shop proprietor, could be viewed as follows



Fig.3.1. Actions of two systems

The two agents each has a range of actions, i.e. the customer can enter the shop, buy a hat and finish by leaving the shop while the shop proprietor can open the shop, sell hats and finishing by closing the shop. If we put these two agents together to form a retail system, at some point in performing the actions open to each, an action of agent Fred's can be to buy and an action of Hatter's can be to sell. We can regard those two actions, of buying and selling, as complementary actions. At the point where these two complementary actions synchronise, there is communication between the two agents. In the diagram which follows, this communication has been shown by the dotted line joining the two complementary actions of buying and selling. Milner [1980] terms this process of combining agents to form one system, a composition operation. We will enlarge on the actual process in a subsequent section.

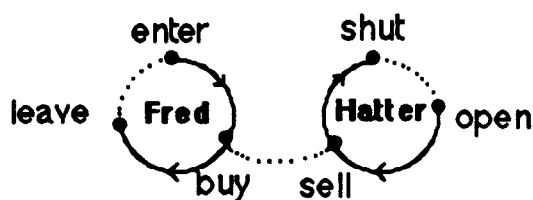


Fig.3.2. Two agents combined

Having performed a composition operation on them, the two systems then form a composite machine, in the case of Fred and Hatter a microcosm of the retailing world, which allows synchronised communication between the complementary actions of its agents. We could equally describe the combination of the above agents more abstractly as follows, in this case the letters a and a^- representing the complementary actions of the machines F and H :

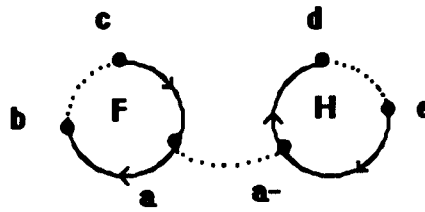


Fig.3.3. A more abstract representation

We now go on to look at the second idea central to the calculus, observational equivalence, again using our miniature retail system as illustration.

3.3.2. Observational equivalence

Intrinsically related to the first idea is that the behaviours, or actions, of a system can be determined by observation, thus a comparison of systems can be made by observing their behaviours. If the observable behaviours of those systems are indistinguishable, they can be defined as observationally equivalent. As this implies, not all the behaviours of a system need be visible, certain actions may be unobservable. Thus the observable behaviour of two systems may be equivalent although one undertakes actions which the other does not. Imagine, for example, that we are standing on the pavement opposite Hatter's establishment observing the actions of Fred. Some of Fred's actions are observable to us and some are hidden from us. The communication action between Fred and Hatter as they perform the complementary actions of buying and selling would, for example, be unobserved by us. Milner calls these communications between complementary actions 'silent' communications, since they are not seen by the observer. This is the basis of the second idea, that of observational equivalence. It is possible, for example, that having in the morning observed Fred enter and later exit the hatshop, that again in the afternoon, from the same vantage point, we see him enter and leave the hatshop for a second time. From

an observational point of view, Fred's shopping behaviour in the morning is equivalent to his shopping behaviour in the afternoon. To us as observers, on each occasion the silent communication action due to the synchronisation of the complementary actions of buying and selling, may or may not have taken place.

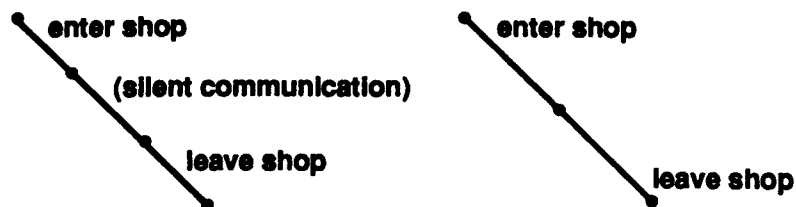


Fig.3.4. equivalent behaviours

Applied to a programming language both the ideas outlined above are potentially very interesting in relation to student modelling. Using them we can think of a program as a set of linked communicating agents, or components. The computation of the silent communications which take place as a result of the composition of the components of a program can be exploited to produce an exact picture of program behaviours, incorrect as well as correct, while the concept of 'observational equivalence' offers the potential of comparing equivalences of programs and parts of programs. The potential of this latter concept for tutoring systems is significant since ultimately it offers the possibility of comparing equivalences of programs and parts of programs.

After this brief overview of the ideas of CCS, we now look more closely at how they are applied. As we have seen, the descriptions derived from the composition of the set of agents which form the system, provide the terms used in the calculus for reasoning about that system and for determining equivalences between systems. For example, from the process of combining the two agents Fred and Hatter to form a

system, we then had a larger set of possible actions resulting from that combination. Using that larger set, we were able to describe two possible sequences of actions which we could say were observationally equivalent. We did not at that point however, enlarge on the actual combination process. When we looked at the 'silent' communication between Fred and Hatter, made possible by the complementary actions of buying and selling, we ignored all the other possible sequences of actions which could have taken place as a result of combining the two agents. From the result of composing the two agents we could have derived a variety of possible sequences of actions or behaviours. We could have expressed all those behaviours by applying Milner's 'expansion theorem', that is, by rewriting the terms of that composition as the sum of all the possible sequences of actions we could derive from that composition. The processes of combining the agents, i.e. the composition operation, and subsequently deriving the sequences of actions made possible by that operation, i.e. applying the 'expansion theorem' [Milner 1980], are basic to the calculus and merit a closer description. Prior to this we should mention one feature which has not yet been mentioned, that non-deterministic choice of actions can also be represented in the system. If for instance we had made a small adjustment to our hat shop scenario and widened the range of goods sold, so that not only hats may be purchased but also cravats, then agent Fred's range of actions could be seen as follows:

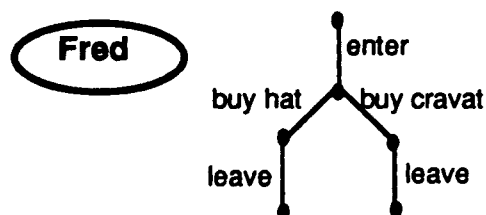


Fig.3.5. a nondeterministic choice of actions - Fred

and the range of Hatter's actions could be seen as follows

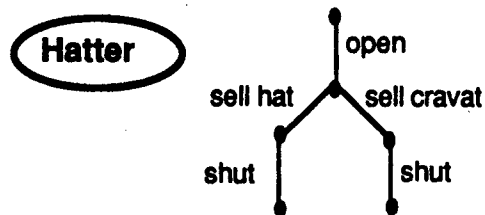


Fig.3.6. a nondeterministic choice of actions - Hatter

Having seen that a choice of actions can be expressed in the system, we now look in more detail at some basic operations of the calculus, using as illustration a system made up of two agents, f and h .

The actions of agent f are an a action or a b action.

$$\text{so } f = (a. f_1 + b. f_2)$$

(where the $+$ symbol represents a nondeterministic choice of actions)

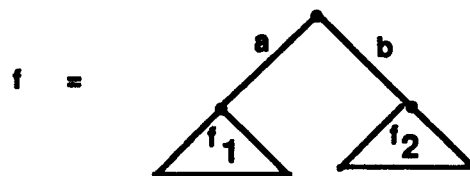


Fig.3.7. agent f

i.e. the agent f can perform the action a , followed by the actions which remain to be performed (any other actions of f that are left) at that state of the tree f_1 , or can perform the action b followed by the actions which remain to be performed (any other actions of f that are left) at that state of the tree f_2 . In this particular case, both f_1 and f_2 are equivalent to nil, which denotes inaction, since at each of those states, f has no remaining actions.

The action of agent h is a -,

$$\text{so } h = (a-. h_1)$$

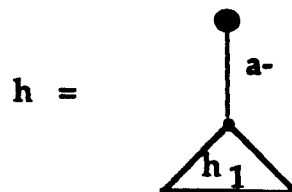


Fig.3.8. agent h

i.e. the agent h can perform the action a - followed by the actions which remain to be performed (any other actions of h that are left) at that state of the tree h_1 , which again in this particular case is equivalent to nil since h has no remaining actions.

The bar symbol - of a - is used to denote that it is a complementary action, in this case complementary to a . As the complementary actions of buying and selling in our hat shop could produce a silent communication when the agents are composed, so the two complementary actions a and a - can also produce a silent communication when these two agents are composed.

3.3.3. Composition

Having defined our two machines, f and h in terms of their actions, we now perform the operation of composing them. The result is a composite machine $f | h$, shown below, in which the choice of possible actions offered by the combined agents is represented by the branches of the composite tree. In [Milner 1980] these diagrams are referred to as synchronisation trees, since they serve to represent the synchronised communications between the agents of the system.

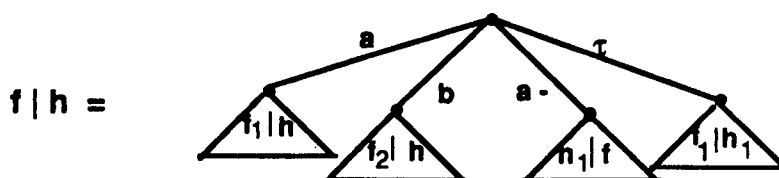


Fig.3.9. composition of agents f and h

This composite machine now offers a choice of actions sequences such that:

- the first action of f , i.e. a or b , can be followed by the composition of the remaining actions of f with h , i.e. respectively, f_1 with h , or f_2 with h .

- or likewise the first action of h , i.e. a^- , can be followed by the composition of the remaining actions of h with f , i.e. h_1 with f .
- or in this case the composite machine also offers the choice of a silent action, because of communication between the complementary actions of a and a^- followed by the composition of f_1 with h_1 (which is nil, since in this case both f_1 and h_1 are nil).

In the diagram above (fig.3.9) we have denoted this silent communication by the branch of the composite tree labelled tau. A more formal definition of composition, taken from [Milner 1980], is shown below.

$$\text{If } f = \sum_i \mu_i f_i \quad \text{and} \quad h = \sum_j \nu_j h_j, \quad \text{then}$$

$$f | h = \sum_i \mu_i (f_i | h) + \sum_j \nu_j (f | h_j) + \sum_{\mu_i = \nu_j} \tau (f_i | h_j)$$

Fig.3.10. definition of composition

So far we have built a description of this system $f | h$ by describing the actions or choice of actions of each agent f and h . We then performed a composition operation on these agents to produce the choice of possible action sequences which the composite system $f | h$ can offer.

3.3.4. Applying the expansion theorem

As we said earlier, this process of constructing a description of the system(s) provides the terms of the calculus. By imposing on these a schema of derivation rules the level of equivalence between systems can be determined. Central to the calculation rules is Milner's expansion theorem. Applied to the result of the composition of a set of agents to form a system, this rewrites the composed agents as a sum of the sum of the actions of each agent. This produces what can be seen intuitively as a tree of all the possible behaviours or actions of the

system. Taking the agents f and h , which we composed to produce the composite machine $f | h$, if we apply the expansion theorem to this, it unfolds, as it were, the complete set of possible sequences of actions for that composite machine, describing each possible sequence of actions at each state of the tree.

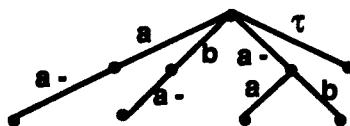


Fig.3.11. composite machine $f | h$ expanded

i.e. $f | h = (+$ (a(a-))
 (b (a-))
 (a- (+ (a) (b)))
 (τ))

The list notation used above to show the results of unfolding the choice of action sequences, is the one we have adopted in the computer implementation of the expansion theorem, which will be described in the following chapter. It should be read as

"the composite machine $f | h$ offers the choice of either:

an a action followed by an $a-$ action

or a b action followed by an $a-$ action

or an $a-$ action followed by
 either

an a action
 or a b action

or a τ action (a silent communication action)"

The expansion theorem producing the tree above is described formally by Milner [1980] p.31 as follows .

Given a synchronization tree 't', expressed as

$$t = \sum_{(1 \leq i \leq n)} \mu_i t_i \quad \text{i.e. the sum of all actions at each state of that tree,}$$

let 't' = $(t_1 | t_2 | \dots | t_m) \setminus A$, where each t_i is a sum as above and A represents a set of action names

then

$$t = \sum \{ \mu ((t_1 | \dots | t_i' | \dots | t_m) \setminus A); \quad 1 \leq i \leq m, \quad \mu t_i' \text{ a summand of } t_i, (\mu) \in A \}$$

$$+ \sum \{ \tau ((t_1 | \dots | t_i' | \dots | t_j' | \dots | t_m) \setminus A); \quad 1 \leq i < j \leq m, \lambda t_i' \text{ a summand of } t_i, \lambda - t_j' \text{ a summand of } t_j \}$$

(λ represents a set of actions,
 $\lambda -$ represents the set of actions complementary to λ)

Fig.3.12. definition of expansion

3.3.5. Restriction

Applying the expansion theorem allows us to rewrite the composition of the agents which go to make up a system. If, however, we are not interested in all the possible sequences of actions, but only interested in sequences which contain or start with certain actions, we can make use of Milner's 'restriction' operator. This allows us to hide the actions which are not of current interest. This operation can be thought of as a pruning one, in that it allows us to specify that certain actions in a system can be considered as hidden behaviours. The ability to consider specific parts of an overall system while ignoring others is a useful one as Zislis [1975] points out in his work on the semantic decomposition of programs and as does Weiser [1981] in his work in Pascal on 'program slicing'. In terms of describing a program the restriction operation allows us to be selective since we can then choose which set of program behaviours we wish to observe. Restriction is represented in Milner

[1980] as a postfix operation and restriction of a particular action or actions applies in every case to the complementary action(s) as well, e.g. $f|h \setminus a$ would restrict all a and all a^- actions in the composition of the machines f and h . Similarly, $f|h \setminus b$ would restrict all b and b^- actions in the composition of f and h . If for instance, in the composition of the machine $f|h$, we had restricted the observable actions so that there would be no occurrences of a or a^- in the results, then instead of the composite machine we originally had as a result of composing f and h , i.e.

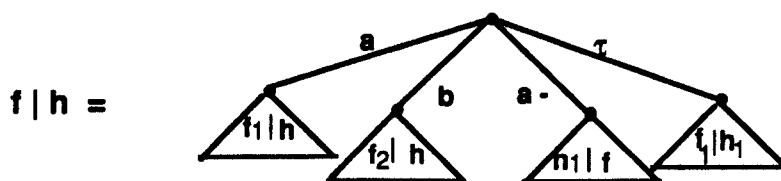


Fig.3.13. machine $f|h$

we would then have the restricted composite machine,

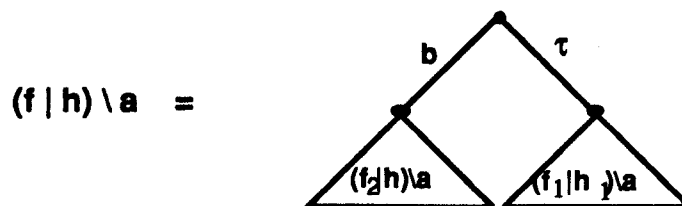


Fig.3.14. restricted machine $f|h \setminus a$

which would in fact only produce two possible sequences of actions, either a b action or a silent communication action, since the composition of $f_2|h \setminus a$ would produce nil and so would the composition of $f_1|h_1 \setminus a$.

A more formal description of this operation on machine f is shown overleaf:

$$\text{for } f = \sum_i \mu_i f_i \text{ we then have}$$

$$f \setminus a = \sum_i \mu_i (f_i \setminus a)$$

$$\mu_i \in (a, a^-)$$

Fig.3.15. a formal definition of restriction

3.3.6. Equivalence

As we stated earlier, applying the expansion theorem to the composition of the agents which make up a system is used to derive all possible behaviours or actions of the composed agents. The terms derived from this rewriting of the action sequences are used in the calculus to prove the equivalence of systems. In comparing the actions of systems, or of parts of systems, Milner [1980] puts forward four levels of equivalence: direct equivalence, strong congruence, observational congruence and observational equivalence, in decreasing order of strength. The notation used to represent these equivalence levels is illustrated below:

equivalence level	denoted by
direct equivalence	\equiv
strong congruence	\sim
observational congruence	\approx^c
observational equivalence	\approx
$B \equiv C \text{ implies } B \sim C \text{ implies } B \approx^c C \text{ implies } B \approx C$	

Fig.3.16. Equivalence levels

Direct equivalence in CCS is a strong relation in that it requires the actions of directly equivalent processes be identical. Strong congruence is also a strong relation, in which in all contexts in the systems being

compared, processes must resemble each other both in their observable behaviour and in the structure of their hidden behaviour. Milner does, however, offer through his calculus a way of formulating definitions of weaker equivalence, in which aspects of the hidden behaviour of systems are disregarded, giving the concept of 'observational' equivalence discussed earlier. Since it is this level of equivalence which is potentially most interesting to us in the context of tutoring systems, we will look at how Milner defines observational equivalence and give a few examples to illustrate this.

We have seen that an agent, for instance agent f , can offer, or perform an action, or range of actions, or communications. If it does so, we can describe this as

- agent f performing certain actions and subsequently behaving as f'
- or
- agent f undergoing certain communication events and subsequently behaving as f' . This is defined as a binary relation over the agent, i.e.

$$f \xrightarrow{\text{action}(s)} f'$$

Fig.3.17. a binary relation over f

The action(s) could also be unobservable, as in our hatshop scenario, where complementary actions can give rise to a silent communication between agent Fred and agent Hatter. The silent communication is represented here by a tau, i.e.

$$f \xrightarrow{\tau} f'$$

Fig.3.18. a silent communication action

Observational equivalence is defined in the following way. If two agents perform the same set of observable actions and their subsequent behaviours are equivalent, then they are observationally equivalent.

$$f \xrightarrow{s} f' \qquad g \xrightarrow{s} g'$$

Fig.3.19. observationally equivalent

For example, picture that we introduce Gertrude, another member of the hatshop clientele. From our observation post on the pavement we see Fred and Gertrude each in turn enter the hatshop and each leave the hatshop. They have performed the same observable action initially, (entering the shop) and their subsequent behaviours are equivalent (leaving the shop), so we can class their behaviours as observationally equivalent.

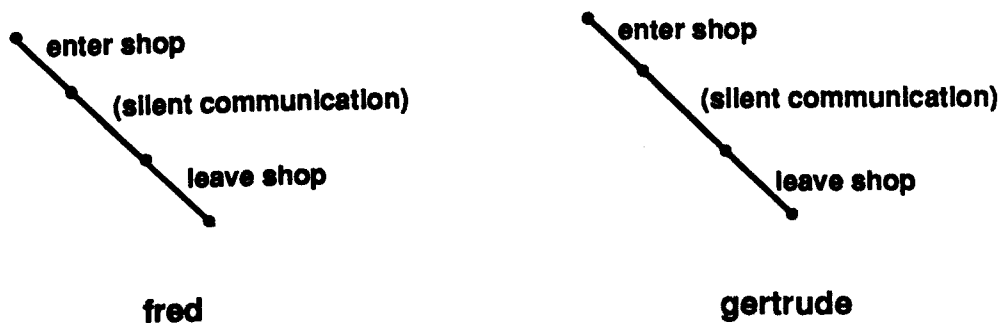


Fig.3.20. observational equivalence of customers' behaviours.

Regardless of whether or not Gertrude purchased a cravat while Fred purchased a top hat, or indeed purchased anything at all, to us as observers, their behaviours were indistinguishable.

A few more examples of pairs of agents which are observationally equivalent are given below to illustrate the concept, preceded by some of the equational axioms of observational equivalence [Milner 1980] against which the examples can be checked.

$$\begin{aligned}
 b &\approx \tau.b \\
 b + \tau.b &\approx \tau.b \\
 a.\tau.b &\approx a.b \\
 a.(b + \tau.c) + a.c &\approx a.(b + \tau.c)
 \end{aligned}$$

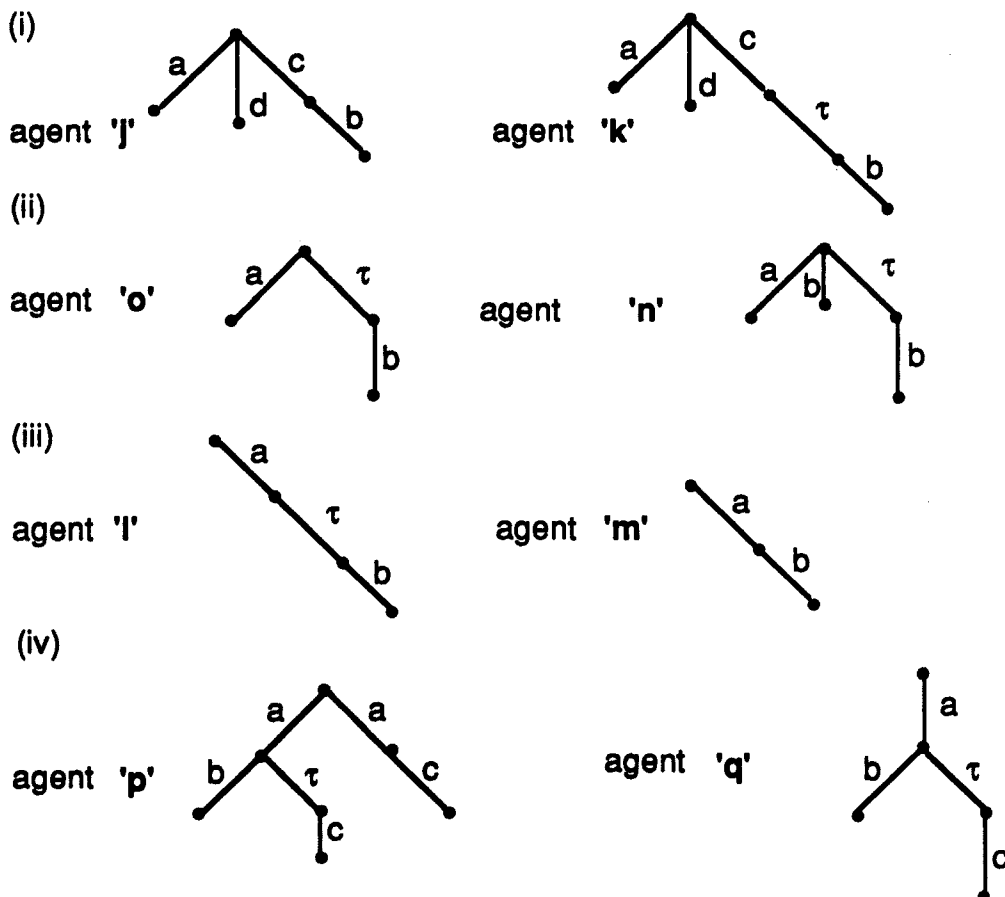


Fig.3.21. examples of observational equivalence

An aspect of CCS which is of particular importance is that as well as having an intuitive interpretation of equivalences which we have put forward here in this chapter, it also offers formal mathematical proofs of the completeness of system descriptions and equivalences. The details of these proofs are not the focus of this work, it being sufficient in this context to note that proofs are based upon accepted mathematical formulae rather than a formalism specific to CCS. Significant contributions to reducing the length of relevant proofs and to producing efficient computer implementations of them have been

made by Milner [1982], Hennessy & Milner [1983], Milner [1985] and by others working on the mathematical foundations of CCS, in particular by Sanderson [1982] and Sanderson [1985]. The exploration of CCS as a formalism in this research focuses principally upon the ideas of the calculus and the diagnostic and explanatory potential which these offer.

3.4. Summary

In this informal description of Milner's calculus, we have outlined the principal ideas of CCS which we feel are relevant to the work in this thesis. We have indicated the relevance of these ideas to diagnostic tutoring, i.e. the possibility of using them to model the behaviours of programs and their promise of providing a means of comparing equivalences of programs or parts of programs. This potential of providing both an exact and accurate semantic description of programs plus the prospect of comparing those programs with a certain flexibility of equivalence levels based upon sound mathematical proofs makes CCS an interesting candidate for exploration in the context of on-line tutoring. In the following chapter we expand on the application of its ideas to Prolog and to formalising novices' misconceptions of the Prolog interpreter which were discussed in the previous chapter.

Chapter Four

4. A Prolog Application

The task of defining a formal semantics for Prolog is not in itself a novel one [Apt & van Emden 1982], [Allison 1983]. It might be thought that as a logic-based programming language Prolog would benefit from having its own inbuilt semantics, those of logic. This has not proved to be the case for anything other than a pure Prolog, i.e. Prolog as a non-deterministic logic programming language. As Fitting [1985] points out, control is not taken into account, although in practice issues of control are often vital, the statement order of a program making the difference between output and no output. In most practical applications Prolog often entails the inclusion of extra-logical features which are implementation dependent. This has led to attempts to formulate a semantics which will encompass such features as the cut [Jones & Mycroft 1984] and other non-logical features such as assert and retract [North 1986]. This work has been undertaken primarily with the goal of checking the correctness of implementations. The focus of our interest in exploring a formal semantics for Prolog has been to produce a formal description of the underlying execution of Prolog. This formal description supports an intuitive procedural interpretation which is used in diagnosing novices' control flow errors. Milner's CCS is designed to handle both synchronisation and value passing in concurrent systems, but to allow a meaningful discussion of backtracking errors we need only use programs drawn from a variable free subset of Prolog. In our exploration of the use of CCS we have therefore restricted ourselves to a subset of Milner's [1980] complete calculus, which involves synchronisation but does not require value-passing.

As was outlined in chapter two, the area chosen for this exploration of a formal semantics, is that of the search and backtracking processes of the Prolog interpreter. An approach to modelling these processes which has been previously tried, is the use of meta-interpreters [Coombs and Stell 1985]. Using CCS however we can produce a precise description of all the execution behaviours which may occur from given Prolog code, rather than a pre-determined selection of those behaviours. We see this as offering a potential flexibility of diagnosis which the implementation of a meta-interpreter would not provide.

The system implemented in this research does not at present deal with errors involving the use of the cut or unification. An extension of the present system would, however, as discussed in chapter seven, necessarily incorporate these aspects of control flow. In this section we initially outline the steps of developing the ideas of CCS in relation to a subset of Prolog. In the remainder of this chapter we then look in more detail at the process of generating semantic descriptions of Prolog programs. The step of incorporating these descriptions in the construction of computational models of those novices' misconceptions of the Prolog interpreter which were discussed in chapter two is postponed until chapter five.

4.1. System overview

The formalism of CCS provides a means of constructing a precise and accurate representation of the possible behaviours of a given system, derived from observation of that system in operation. To relate our previous overview of CCS to programming languages, we must then consider a program as a process comprising a certain number of components, each component itself being a process. Thus observing a

program in this context consists of observing the actions and interactions of the combined processes in operation, since each component process is able to execute certain actions and in doing so may interact, or communicate, with the other processes. By an expansion operation, we examine in more detail what those interactions may be for any given program, i.e. we unpack the actions or behaviours of the combined component processes. This combination and expansion produces, as it were, a parallel semantics of a given program, that is, it allows us to represent the parallel execution of each program. In a Prolog program this representation includes not only 'correct' backtracking behaviours, but also 'incorrect' backtracking behaviours. We are not however, necessarily interested in all the actions and interactions or communications of a particular process, so we make use of the restriction operation, which effectively allows us to stipulate which actions or communications we wish to be concealed, or hidden, and which interactions we wish to examine. It is from this expansion of selected interactions of the component processes that we extract the semantic representation of a given program which traces all the possible execution paths of that program. By using this representation in conjunction with a production rule system, we then construct and verify the models of backtracking behaviours to be used in diagnosis.

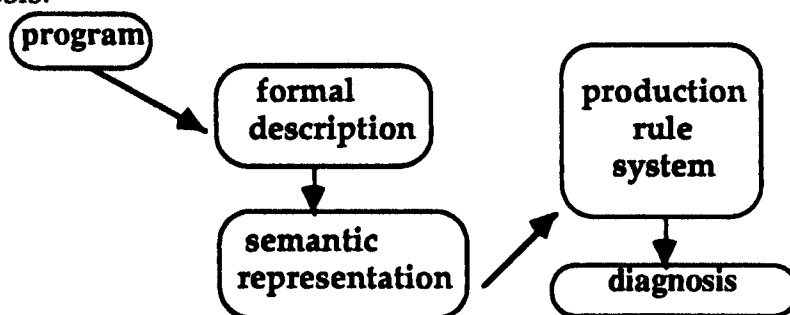


Fig.4.1. Process outline

The diagrammatic outline of the overall process given above (fig.4.1), illustrates the relationships between the sections of work which are discussed in this and the following chapter. As can be seen from that diagram, initially we produce a formal description of the Prolog program being used. The results of this process are subsequently employed to generate its semantic representation. This semantic representation is then used within the framework of a production rule system to construct models of students' misconceptions of the Prolog interpreter to be used in diagnosis. We now look in more detail at each part of the overall process outlined above.

4.2. Producing formal descriptions of Prolog programs

Formally describing the actions of each component of a Prolog program entails defining what constitutes its components, just as we defined the agents Fred and Hatter as being the components of our miniature retail system in chapter three.

In doing this for Prolog programs our purpose has been to capture their most general features. We have been influenced to a certain extent by the goal of our current work of diagnosing backtracking errors. The definition of a program's components which we have arrived at is therefore not necessarily the only one possible, but rather one which seemed most appropriate for the task.

We have defined a Prolog program as being made up of some or all of the following components: query elements, condition elements, right hand side elements and fact elements. Take for example the simple program:

```
p if a.
a.
```

As can be seen in fig.4.2 below we have defined this program as containing two query components, a conditional component and a fact component. In the diagram, 'lhs-if-rhs' represents the conditional element of the program (that the left hand side of the program is true if the right hand side is true). The right hand side is represented by 'rhands', which in this case has only one element, 'a'. Each component and its semantic function will be discussed in detail in sections.[4.2.1] to [4.2.5].

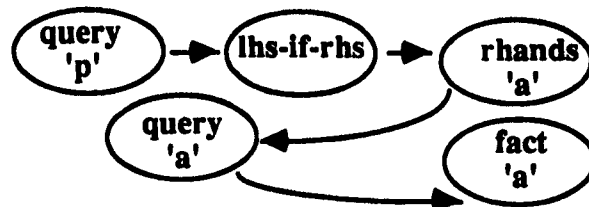


Fig.4.2 Formulating components

In addition to these components, we also include 'nomore' elements. The purpose of this is to express the 'closed world' assumption of Prolog explicitly and to allow the system to exhibit the full range of backtracking possibilities for each program. Thus the program above, with the addition of 'nomore p' and 'nomore a' would be fully defined as having seven components.

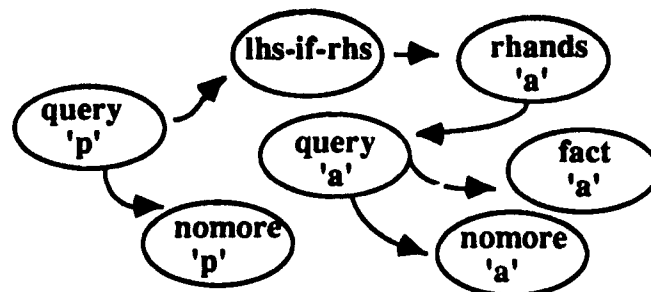


Fig.4.3 Components of program

The components are represented below in the format used as input for the program which produces the description of each machine's possible actions.

```

((query "P" 2)
 (lhs-if-rhs "P" 1)
 (rhands "P" 1 "A")
 (query "A" 2)
 (fact "A" 1)
 (nomore "P" 2)
 (nomore "A" 2))

```

As with the agents or machines referred to in the previous chapter, each component process is capable of a certain sequence or sequences of actions. Just as we then combined the actions of the agents Fred and Hatter by making use of the CCS composition operation, these seven 'agents' or 'machines' of the program can be combined and can interact or communicate with each other through their complementary actions. In the following sections, we look at the components in turn and show the action sequences which we have assigned to each.

4.2.1. Query components

Taking the head of the list first (although the ordering is not in any way significant to the results), we look in more detail at each component, the actions of which it is capable and explain the syntax used in converting it into a CCS machine.

The first element of the three element list (query "P" 2), is a component identifier. The second element is the name of the component and the third element is its clause number. In the case of query components the clause number is always one higher than the actual number of clauses present with this particular name. This, working together with 'nomore' components, is to satisfy the objective of showing all possible backtracking behaviours.

The initial action of a query component is to contact an instance of a machine of that name, which in the case of a query to p , is a p clause. This could result in a successful communication or an unsuccessful

communication, i.e. a call to p , starting the process in motion and which we represent as (SP), would instigate contact with a machine (SP1). The numeral '1' serves to distinguish this as the first p clause of possibly multiple p clauses which might be contacted in a given program. The result of this action is either that (SP1) is contacted successfully, in which case (SP) performs its final action successfully, as (SP-), or it is not, in which case the failure of (SP) is denoted by a final action (FP-). As there is only one p clause, the syntax of our query machine for p would look, so far, as follows (the '+' symbol representing non-deterministic choice):

```
(SP (SP1- (+
            (SP1 (SP-))
            (FP1 (FP-)))))
```

This syntax can be read as 'this machine is able to offer the action SP followed by the action SP1-, followed by either the action SP1 followed by the action SP-, or the action FP1 followed by the action FP-'. However, the extra clause to each query, included to represent explicitly the closed world assumption in Prolog and to produce all the possible backtracking behaviours, means that the call to p can also instigate contact with a second p clause (SP2-).

This action too may lead to a successful or unsuccessful communication, correspondingly performing either a successful final action (SP-) or an unsuccessful (FP-). Thus a function call with the input (query "P" 2) would produce the following description of that component's actions as output:

```
(SP (+
     (SP1- (+
            (SP1 (SP-))
            (FP1 (FP-)))))
     (SP2- (+
            (SP2 (SP-))
            (FP2 (FP-)))))
```

So our 'query p' component of fig.4.3, seen with its actions or behaviours explicitly marked, starting from 'SP' would look as follows:

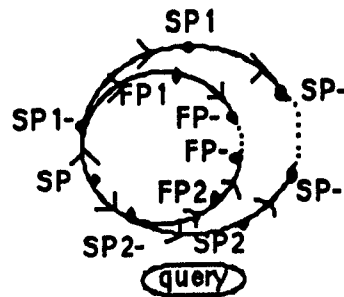


Fig.4.4 Actions of query component

Likewise, if we were to convert the program:

**breakfast if eggs.
eggs.**

the same list of components would be applicable and the conversion of the query component, i.e.

(query "BREAKFAST" 2)

would similarly, in the syntax we have used for the implementation, produce the following description of the machine's actions:

```
(SBREAKFAST(+
  (SBREAKFAST1- (+
    (SBREAKFAST1 (SBREAKFAST- ))
    (FBREAKFAST1 (FBREAKFAST- )))
  (SBREAKFAST2- (+
    (SBREAKFAST2 (SBREAKFAST- ))
    (FBREAKFAST2 (FBREAKFAST- ))))))
```

while given the program:

**breakfast if eggs.
breakfast if bacon.
eggs.**

the result of converting the query machine to produce a description of its actions would reflect the additional 'breakfast' clause, so

(query "BREAKFAST" 3)

shows the actions of that machine as:


```

(SBREAKFAST (+
  (
    (SBREAKFAST1- (+
      (SBREAKFAST1 (SBREAKFAST-))
      (FBREAKFAST1 (FBREAKFAST-))))
    (SBREAKFAST2- (+
      (SBREAKFAST2 (SBREAKFAST-))
      (FBREAKFAST2 (FBREAKFAST-))))
    (SBREAKFAST3- (+
      (SBREAKFAST3 (SBREAKFAST-))
      (FBREAKFAST3 (FBREAKFAST-))))))

```

As shown, this component has the possibility of contacting both the 'breakfast' clauses available and a third possibility of contact which would allow for contact with the relevant 'nomore breakfast 3' machine.

4.2.2. The condition component

This component (lhs-if-rhs "P" 1) expresses the condition that the left hand side of the program is true if the right hand side is true. Again represented as a three element list, the first element is the component identifier, the second the name of the left hand side and the third element the number of the clause concerned. The actions of this machine consist of the left hand side clause contacting its right hand side. In our example program *p* if *a*, this would be represented as the first left hand clause of *p*, (SP1) contacting its right hand side (SP1RHS-). This may be a successful communication (SP1RHS), in which case the first clause *p* of the left hand side is successful (SP1-) or it may be an unsuccessful communication (FP1RHS), in which case the first clause of *p* of the left hand side is unsuccessful (FP1-). So, inputting the component (lhs-if-rhs "P" 1) would produce the following description of this machine's actions:

```

(SP1 (SP1RHS- (+
  (SP1RHS (SP1-))
  (FP1RHS (FP1-))))

```


the right hand side would be an unsuccessful one (FA (FP1RHS-)).
 Converting the component (rhands "P" 1 "A") would thus produce the
 following description of the actions of that machine:

(SP1RHS (SA- (+
 (SA (SP1RHS-)
 (FA (FP1RHS-))))))

Thus the rhands component of fig.4.3 with its actions explicitly stated
 can be represented as follows:

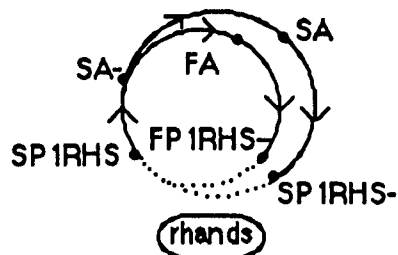


Fig.4.6 Actions of the right hand side

If we are converting a program which has more than one condition as
 its right hand side, for example the program:

'p if a & b'.

or the program:

'breakfast if eggs & bacon.'

the conversion of this component to produce a description of the
 actions of that component would in doing so reflect the logical
 conjunction. In these cases the right hand side is only successful if
 successful contact is made with all the relevant machines, which in the
 first program would be both a and b, in the second both eggs and bacon.
 This is expressed in terms of a parallel 'and', allowing for
 communicative actions in any order, again, in order to capture
 'incorrect' as well as 'correct' behaviours of the program. Let us take for
 example, the machine (rhands "P" 1 "A" "B") which would be a

component of the program *p* if *a* & *b*. The actions of the machine produced from this component would allow for successful contact of the right hand side,

- either if successful contact is made with the *a* component followed by successful contact with the *b* component,
- or if successful contact is made first with the '*b*' component followed by successful contact with the '*a*' component.

Reflecting the logical conjunction, at the first unsuccessful contact of the right hand side, the contact offered by this component is then an unsuccessful one, as we can see from the output produced, for example, from converting the component (rhands2 "P" 1 "A" "B") to a description of its actions.

```
(SP1RHS ;contacts
  (+ ;either
    (SA- (+ ;which either
      (SA (SB- ;succeeds, so contacts 'b'
        (+ ; which either
          (SB (SP1RHS- ));succeeds, so rhands succeeds
          (FB (FP1RHS- )))) ;or fails so rhands fails
        (FA (FP1RHS- ))));or fails, so rhands fails
      ;or
      (SB- (+ ;which either
        (SB (SA- ;succeeds, so contacts 'a'
          (+ ;which either
            (SA (SP1RHS-)) ; succeeds, so rhands succeeds
            (FA (FP1RHS-)))) ;or fails, so rhands fails
          (FB (FP1RHS-)))) ;or fails, so rhands fails
```

Similarly, if there are three components to the right hand side, as would be the case if, for instance, we added 'tomatoes' to our breakfast program, making it 'breakfast if eggs & bacon & tomatoes', then from the input: (rhands "BREAKFAST" 1 "EGGS" "BACON" "TOMATOES") we would get the following possible actions from that machine:

```
(SBREAKFAST1RHS(+
  (SEGGs- (+
    (SEGGs (+
      (SBACON- (+
```

```

(SBACON (STOMATOES-
  (+
    (STOMATOES(SBREAKFAST1RHS-))
    (FTOMATOES (FBREAKFAST1RHS-))))))
(FBACON (FBREAKFAST1RHS-)))
(STOMATOES- (+
  (STOMATOES (SBACON- (+
    (SBACON(SBREAKFAST1RHS-))
    (FBACON (FBREAKFAST1RHS-))))))
  (FTOMATOES (FBREAKFAST1RHS-))))))
(FEGGS (FBREAKFAST1RHS-)))
(SBACON- (+
  (SBACON (+
    (SEGGS- (+
      (SEGGS (STOMATOES- (+
        (STOMATOES (SBREAKFAST1RHS-))
        (FTOMATOES(FBREAKFAST1RHS-))))))
      (FEGGS (FBREAKFAST1RHS-))))))
    (STOMATOES- (+
      (STOMATOES (SEGGS- (+
        (SEGGS (SBREAKFAST1RHS-))
        (FEGGS (FBREAKFAST1RHS-))))))
      (FTOMATOES (FBREAKFAST1RHS-))))))
    (FBACON (FBREAKFAST1RHS-))))))
  (STOMATOES- (+
    (STOMATOES (+
      (SEGGS- (+
        (SEGGS (SBACON- (+
          (SBACON (SBREAKFAST1RHS-))
          (FBACON (FBREAKFAST1RHS-))))))
        (FEGGS (FBREAKFAST1RHS-))))))
      (SBACON- (+
        (SBACON (SEGGS- (+
          (SEGGS (SBREAKFAST1RHS-))
          (FEGGS (FBREAKFAST1RHS-))))))
        (FBACON (FBREAKFAST1RHS-))))))
      (FTOMATOES (FBREAKFAST1RHS-))))))
    (STOMATOES (FBREAKFAST1RHS-))))))
  (STOMATOES (FBREAKFAST1RHS-))))))

```

In all cases, at any stage of the enterprise, the order of contact offered is non-deterministic (either eggs, or bacon or tomatoes can be produced in any order), but at the first unsuccessful contact action, (if at any stage we fail to produce one of the ingredients), then the contact offered by that component is an unsuccessful one (the whole breakfast endeavour fails).

It is worth bearing in mind that the actual behaviour of the Prolog interpreter produces only one of those behaviour sequences captured by the three way conjunction.

4.2.4. Fact and nomore components

Both fact and nomore components have more limited actions. Facts are defined as simply having a successful action, while the representation of a fact not being available, a 'nomore' component, has only an unsuccessful action. The call (fact "A" 1) produces the description of this machine's actions:

(SA1(SA1-))

and the input (nomore "A" 2) produces the description:

(SA2(FA2-))

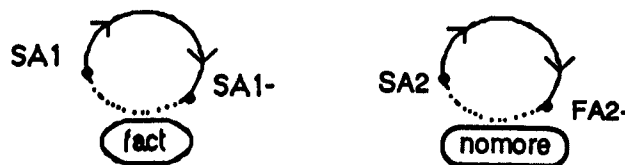


Fig.4.7 Actions of fact and nomore components

Similarly, converting the fact component in the program:

breakfast if eggs.

eggs.

i.e. a call to (fact "EGGS" 1)

produces the following description of that machine's actions:

(SEGGS1 (SEGGS1-))

and similarly, converting the 'fact not available' component, i.e., the 'nomore' component, produces, from the call (nomore "EGGS" 2), the actions:

(SEGGS2 (FEGGS2-))

4.2.5. A program converted

The final outcome then of converting the components of the program:

p if a.

a.

into CCS machines is a list of all the components which comprise the process, described in terms of their actions. i.e.

((SP (+ (SP1- (+ (SP1 (SP-)) (FP1 (FP-)))) (SP2- (+ (SP2 (SP-)) (FP2 (FP-))))))	query component
(SP2 (FP2-))	no more component
(SA (+ (SA1- (+ (SA1 (SA-)) (FA1 (FA-)))) (SA2- (+ (SA2 (SA-)) (FA2(FA-))))))	query component
(SA2 (FA2-))	nomore component
(SP1 (SP1RHS- (+ (SP1RHS (SP1-)) (FP1RHS (FP1-))))	left-hand/right-hand side condition relationship
(SP1RHS (SA- (+ (SA (SP1RHS-)) (FA (FP1RHS-))))	right-hand side component
(SA1 (SA1-))	fact component

We have now specified the possible actions of each machine which comprises the program, i.e. our components of fig.4.3 now have their actions formally described. The code used in implementing this conversion of program components into machines expressing their range of actions is listed in Appendix B1.

Having automated the production of the CCS machine descriptions from the Prolog program components, i.e. formally described the possible channels through which communication can take place between these machines, the next step is to model those interactions.

We do this by composing the machines and applying the expansion theorem to the results. In chapter three, as a result of composing the agents Fred and Hatter to form a system we saw that if we were to express all the possible sequences of actions that were then able to be offered, i.e. applied the expansion theorem to their composition, one action offered was a silent communication between the two agents.

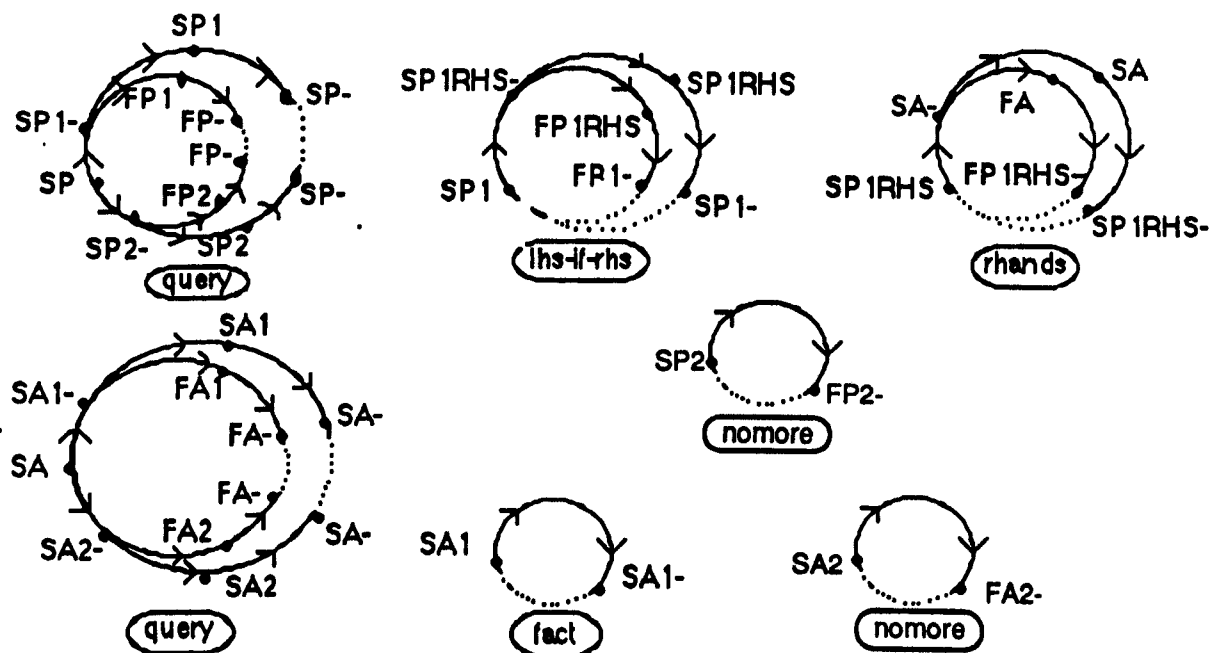


Fig.4.8 Formal description of actions of machines shown earlier in fig.4.3

Similarly, as a result of composition and expansion we can generate all the possible silent communications which can take place between the components of a program, thus representing not only the correct but also incorrect program behaviours. This representation forms the basis of interpreting student input and diagnosing errors. In the following section we look more closely at the process of applying the expansion theorem to the composition of the components of a program.

4.3.Expanding the formal description of a program

The composition of the machines of a process into one system allows communications between those machines. As we discussed in the previous chapter, the expansion theorem [Milner 1980] is a calculation rule which states that such a composition can be rewritten into a sum of the sum of the actions of that system. In this section we briefly consider again the composition operation, the kind of communication between machines which it allows to take place and the use of the expansion theorem in tracing these communications, but in this instance in relation to Prolog programs. We then go on to describe the process of automating the application of the expansion theorem, which allows us to produce a semantic representation of a program's execution in CCS terms, giving some examples as illustration. Code used in implementing the expansion algorithm discussed below is included in Appendix B2.

4.3.1.Composing two program components

We have said that by composing the machines of a process, we are combining their possible actions and possibilities of communication. By then applying the expansion theorem, that is, rewriting the composition of machines into a sum of the possible actions, we are able to see in detail all the action sequences and internal communications which can take place between the machines of a particular system or process. Here, using the components of a Prolog program we show those processes in more detail. Take for instance, the fact component and the nomore component of fig.4.7 [section4.2.3], illustrated here in tree form.

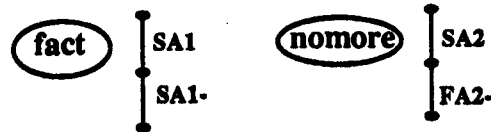


Fig.4.9 Actions of fact component and nomore component

By composing these machines, just as fig.3.9 [section 3.3.3] in the previous chapter showed that the composition of the machines f and h resulted in the composite machine $f|h$, so here the result is the composite machine $\text{fact}|nomore$.

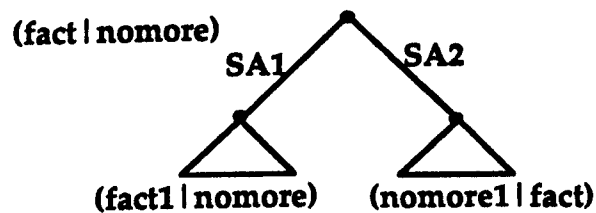


Fig.4.10 composition of fact and nomore machines

and similarly, we now have the possibility of:

- an SA1 action followed by the sequences of actions made possible by the composition of what actions remain from the fact machine (which in this case is SA1-) with the actions of the nomore machine
- or
- an SA2 action followed by the sequences of actions made possible by the composition of what actions remain from the nomore machine (which in this case is the action FA2-) with the actions of the fact machine.

though unlike the composite machine $f|h$, the machine $\text{fact}|nomore$ does not offer a silent communication action.

4.3.2. Applying the expansion theorem

Having composed these components, we can now apply the expansion theorem to the composite machine, to inspect the possible action sequences in detail. As a result we will then have the sum of all the possible sequences of their combined actions, just as we did from the expansion of the composite machine $f|h$ in fig.3.11 [section3.3.4].

is now ((SA1-)(SA2 (FA2-))). Subsequently the first element of the second machine, (SA2(FA2-)), is selected and joined to the results of expanding the remainder of the list ((SA1(SA1-))(FA2-)). If there is more than one machine in the list of machines to be expanded, a plus symbol '+' is inserted as an indication that at that point there are more machines to be expanded, i.e. that there is a choice of action sequences. The position to date would be the choice of actions,

(+ (SA1(followed by the expansion of rest))
(SA2(followed by the expansion of rest))).

In each case, as we can see above, there is still more than one machine in the remaining list of machines to be expanded, so the algorithm is applied again to that list. In the next steps the first element of each machine in that remaining list would be joined to the expansion of any remainder, so the position would subsequently be:

(+ (SA1 (+
(SA1- (followed by expansion of rest))
; 'rest'=one machine, ((SA2(FA2-)))
(SA2 (followed by expansion of rest))))
; 'rest'=two machines, ((SA1-)(FA2-))
(SA2 (+
(SA1 (followed by expansion of rest))
; 'rest'=two machines, ((SA1-)(FA2-))
(FA2- (followed by expansion of rest))))
; 'rest'=one machine, ((SA1(SA1-)))

The algorithm is applied repeatedly until there are no more machines to expand and the process is completed, giving us the result:

(+ (SA1 (+ (SA1- (SA2 (FA2-)))
(SA2 (+ (SA1- (FA2-))
(FA2- (SA1-))))))
(SA2 (+ (SA1 (+ (SA1- (FA2-))
(FA2- (SA1-)))
(FA2- (SA1 (SA1-))))))

This can also be output in tree form since this makes it easier to illustrate all the sequences of actions clearly and intuitively in a

compact layout, in which case the list output is passed on to a tree-drawing algorithm and the results would be much as illustrated above in fig.4.11. As can be seen from that figure, from the expansion of the composite machine `fact|nomore`, we have derived six possible action sequences.

4.3.3. Tagging the silent communications of a program

Having composed and expanded the composite machine to look at the possible sequences of actions, there is then an opportunity to observe any silent communications which can take place between the machines. In our formal descriptions of Prolog programs outlined in this chapter, we denote, as we did in our outline of CCS in chapter three, those actions which are complementary to each other by the use of names and co-names, the co-name being marked by a postfix bar. The actions of (SP) and (SP-), or of (SA) and (SA-), for example, are complementary to each other. In a composition and expansion of machines, if the action offered by one machine synchronises with a complementary action of another machine, a 'silent' communication can take place. To illustrate this in a Prolog context, we take the two machines, the (query "A" 1) component and the (fact "A" 1) component, shown previously in fig.4.8 [section 4.3.4] as components of our `p` if a program and whose actions are as follows:

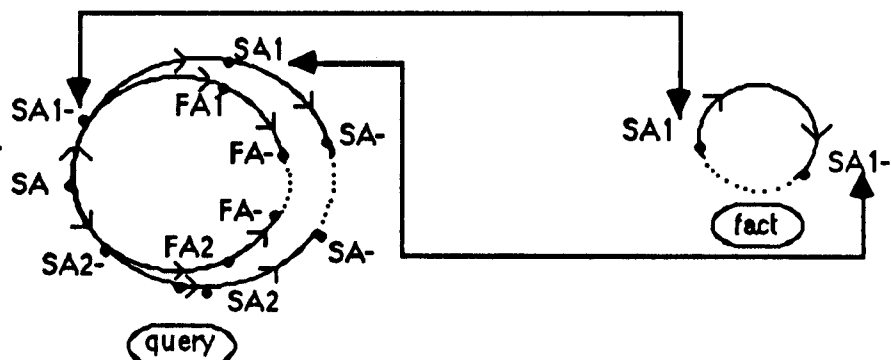


Fig.4.12 possible complementary actions of SA1- and SA1

We can see that in certain sequences of actions, the actions SA1 and SA1- of the query machine, will possibly synchronise with their complementary actions SA1- and SA1 of the fact machine. This allows internal 'silent' communications to take place between the two machines. To illustrate this, in applying the expansion algorithm to the composition of these two machines so that we can examine all the possible sequences of actions which could result, we highlight the sequences in which silent transitions occur. The function

```
(expand-machines '((query "A" 2)(fact "A" 1)
```

would initially convert these components into their CCS machines,

```
((SA (+ (SA1- (+ (SA1 (SA-))
                  (FA1 (FA-))))
        (SA2- (+ (SA2 (SA-))
                  (FA2 (FA-)))))
  (SA1 (SA1-)))
```

then the function 'expand-machines', would proceed with the expansion process:

```
(expand-machines '(*(SA (+ (SA1- (+ (SA1 (SA-)) (FA1 (FA- ))))
                          (SA2- (+ (SA2 (SA-)) (FA2 (FA- )))))
                  (SA1 (SA1- )))nil)
```

giving as result the sum of all the sequences of actions obtainable from that expansion:

```
(+
  (SA (+
    (SA1- (+
      (SA1 (+
        (SA- (SA1 (SA1-)))
        (SA1 (+ (SA- (SA1-)) (SA1- (SA-)))))
      (FA1 (+
        (FA- (SA1 (SA1-)))
        (SA1 (+ (FA- (SA1-)) (SA1- (FA-)))))
      (SA1 (+
        (SA1 (+ (SA- (SA1-)) (SA1- (SA-)))
        (FA1 (+ (FA- (SA1-)) (SA1- (FA-)))
        (SA1- (+ (SA1 (SA-)) (FA1 (FA-)))
        ("τ SA1" (SA-)))))
    (SA2- (+
```

```

(SA2 (+
      (SA- (SA1 (SA1-)))
      (SA1 (+ (SA- (SA1-)) (SA1- (SA-)))))
(FA2 (+
      (FA- (SA1 (SA1-)))
      (SA1 (+ (FA- (SA1-)) (SA1- (FA-)))))
(SA1 (+
      (SA2 (+ (SA- (SA1-)) (SA1- (SA-)))
      (FA2 (+ (FA- (SA1-)) (SA1- (FA-)))
      (SA1- (+ (SA2 (SA-)) (FA2 (FA-)))))
(SA1 (+
  (SA1- (+
        (SA1 (+ (SA- (SA1-)) (SA1- (SA-)))
        (FA1 (+ (FA- (SA1-)) (SA1- (FA-)))
        (SA1- (+ (SA1 (SA-)) (FA1 (FA-)))
        (" $\tau$  SA1" (SA-)))
      (SA2- (+
        (SA2 (+ (SA- (SA1-)) (SA1- (SA-)))
        (FA2 (+ (FA- (SA1-)) (SA1- (FA-)))
        (SA1- (+ (SA2 (SA-)) (FA2 (FA-)))))
      (SA1- (+
        (SA1- (+ (SA1 (SA-)) (FA1 (FA-)))
        (SA2- (+ (SA2 (SA-)) (FA2 (FA-)))))
      (" $\tau$  SA1" (+
        (SA1 (+ (SA- (SA1-)) (SA1- (SA-)))
        (FA1 (+ (FA- (SA1-)) (SA1- (FA-)))
        (SA1- (+ (SA1 (SA-)) (FA1 (FA-)))
        (" $\tau$  SA1" (SA-)))))
(SA1 (+
  (SA (+
    (SA1- (+
      (SA1 (+ (SA- (SA1-)) (SA1- (SA-)))
      (FA1 (+ (FA- (SA1-)) (SA1- (FA-)))
      (SA1- (+ (SA1 (SA-)) (FA1 (FA-)))
      (" $\tau$  SA1" (SA-)))
    (SA2- (+
      (SA2 (+ (SA- (SA1-)) (SA1- (SA-)))
      (FA2 (+ (FA- (SA1-)) (SA1- (FA-)))
      (SA1- (+ (SA2 (SA-)) (FA2 (FA-)))))
    (SA1- (+
      (SA1- (+ (SA1 (SA-)) (FA1 (FA-)))
      (SA2- (+ (SA2 (SA-)) (FA2 (FA-)))))
    (SA1- (SA (+
      (SA1- (+ (SA1 (SA-)) (FA1 (FA-)))
      (SA2- (+ (SA2 (SA-)) (FA2 (FA-)))))

```

It is these 'silent' communications, highlighted above, in which we are primarily interested, since they show the channels of communications between the components which provide the possible execution paths of a program. In implementing the expansion theorem, we have tagged the silent communications (which as in chapter three, are represented

by a tau symbol) with the action which gave rise to it. As can be seen in the above results, in this case it was on each occasion either the action 'SA1' synchronising with the complementary action 'SA1-' or the action 'SA1-' synchronising with the complementary action 'SA1'. The purpose of this tagging is to identify which communications between components have led from the start of the program to either a successful or unsuccessful outcome.

4.3.4. Restricted observation of a program

The results shown above also serve to illustrate the potential use of the restriction operation which we discussed in chapter three [section3.3.5]. As we indicated then, it allows us to limit the appearance in the results of action sequences in which we are not interested. In expanding the query and fact components above, we were basically only interested in observing the silent communications, not the large number of other possible action sequences. We then use the restriction operation to hide all those other action sequences occurring between the start of the execution process and its finish. In this case the results of expanding the two components above, are then observed as follows,

```
(SA
  ("τ SA1-"
   ("τ SA1"
    (SA-))))
```

allowing us only the view of program execution in which we were interested, i.e. the successful communication sequence leading from the start action of the machine query "A", via the silent communications that took place between the two components and ending with the successful state of machine fact "A".

As an example of another use of the restriction operator in the process of expanding components of a program, we could look again at the expansion of the machines in fig.4.9 above [section 4.3.1]. These are a fact component and a nomore component and the results of their expansion are shown in fig.4.10 of the same section. If, for instance, instead of showing all the possible action sequences of that expansion, we wished to hide all occurrences of the action 'FA2-' in the results, we would specify at the outset of the process that the action 'FA2-' is to be restricted. In our implementation of the expansion theorem, this entails adding the name of the restricted action into the list which is the second argument to the function 'expand-machines'. In our earlier example of a call to 'expand-machines' [section 4.3.2] this had 'nil' as its second argument, since at that point we did not wish to restrict any of the actions from appearing in the result.

(expand-machines '(*(SA1 (SA1-))(SA2 (FA2-))) nil)

Now, in order to express this restriction of 'FA2-', the call to the expansion function would be:

(expand-machines '(*(SA1(SA1-))(SA2(FA2-))) '(FA2-))

As the algorithm is implemented, a check is made on each element of the machines to see if it is a member of this list of restricted actions. If it is, that element is treated as nil, i.e. it is not added as an action in the results, nor are the actions which follow it in the machine where it occurs. So having restricted all occurrences of FA2- in our example, the results will show only the following action sequences.

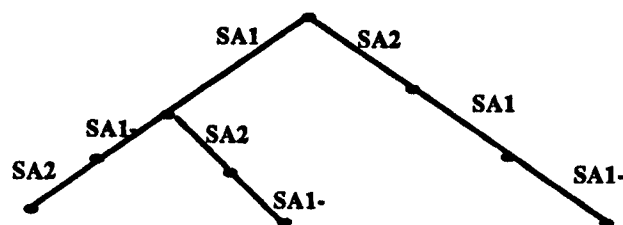


Fig.4.13 restricted expansion of (SA1(SA1-)) (SA2(FA2-))

This should be compared with the results in fig.4.10, in which no actions were restricted. It can be seen from this example that the restriction operator is also a useful counterbalance to the possibility of combinatorial explosion.

4.4. Summary

In the preceding sections we have shown how by applying the ideas of CCS to Prolog, we can generate a semantic description of a Prolog program. We have described a program in terms of the behaviours of the set of components which combine to form the system. The composition and expansion of these components show us all the possible sequences of actions and silent communications which can be derived from that program. The use of the restriction operator allows us to hide the sequences of actions in which we are not currently interested and to show only the silent communications. These silent communications represent the possible communication paths between the different components of the system. An example of this, given below, is the set of possible communication paths for the program we discussed earlier:

p if a.
a.

This program contained seven components (fig.4.8). These were composed and expanded, restricting all actions between the start and finish states except for the silent communications, which are tagged to show which components gave rise to them. The result is a description of the silent communications which can take place between the components of that program from the start state to possible finish states.

These silent communications represent the possible flow of control paths which could take place in the execution of that program.

```
(SP (+
  ("τ SP1-"("τ SP1RHS-"("τ SA-"(+
    ("τ SA1-"("τ SA1"("τ SA"("τ SP1RHS"("τ SP1"(SP-))))))
    ("τ SA2-"("τ FA2"("τ FA"("τ FP1RHS"("τ FP1"(FP-)))))))))
  ("τ SP2-"("τ FP2"(FP-))))))
```

It is intuitively easier to see these paths if the results are represented in tree form, so this output is also passed to a tree drawing algorithm. This gives us the following result:

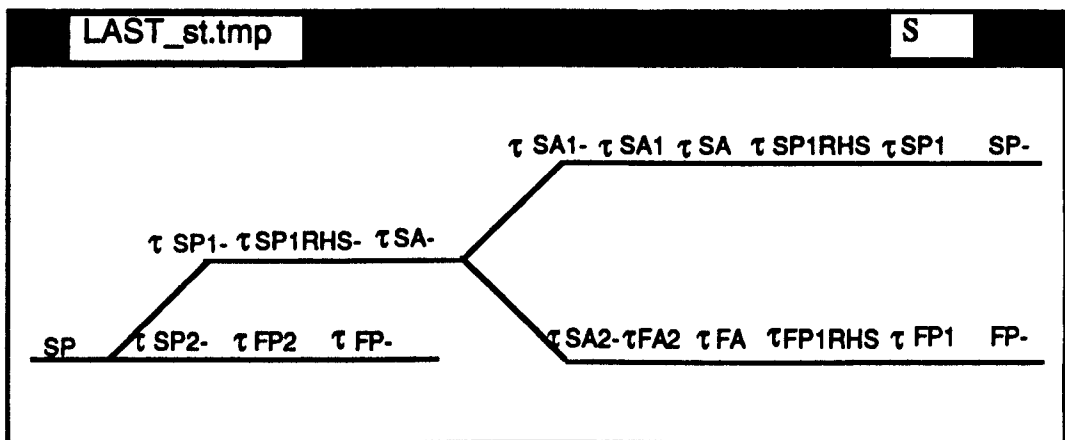


Fig.4.14 CCS tree representation of program 'p if a'. 'a'.

This semantic representation of the Prolog program is then used as the basis for constructing models of the Prolog interpreter employed in the diagnosis of students' misconceptions. This representation can be seen as the space of possible execution paths for that program. Modelling the paths through that space is undertaken within the framework of a production rule system. In the following chapter we look at the development of this system and its production of models of the Prolog interpreter.

Chapter Five

5. Production rule modelling

As we have seen in the previous chapter, the semantic representation of possible execution paths of a Prolog program can intuitively be thought of as a CCS search tree. In this chapter we discuss our use of these representations in diagnosing novices' misconceptions and describe their incorporation in a production rule system designed to undertake this diagnosis. In considering our choice of a production rule approach, we outline the questions raised in developing such a system and the way we have attempted to answer these. It should be noted that the motivation for adopting a production rule approach was based purely on the practical advantages of such a system. The rules which have been developed to model the execution of Prolog programs are not intended to be interpreted as having psychological validity. The benefits which are to be gained from using a production rule interpreter and which therefore motivated this approach are discussed below

5.1. A production rule approach

It will be helpful in the ensuing discussion if we consider a particular Prolog program as a point of reference, so for this purpose we shall take the following program:

```
p if a & b.  
a.  
a.
```

Generating the semantic representation of this program and passing the results as input to the tree-drawing algorithm gives the following tree, shown overleaf in fig.5.1.

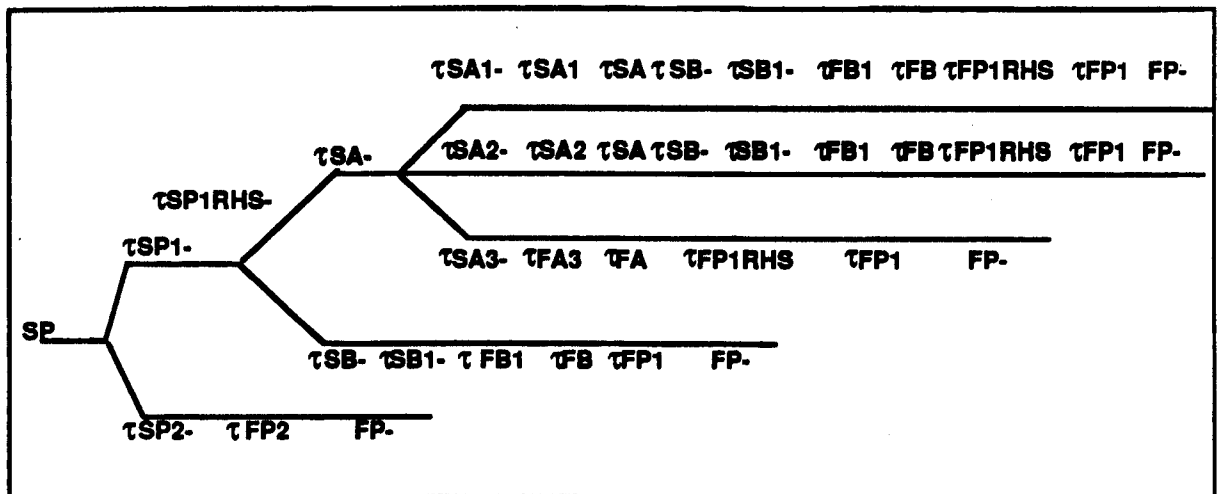


Fig.5.1 CCS tree of p if a & b.

Modelling correct and incorrect execution paths can be seen as selecting particular paths represented by certain branches of that tree. From the example given above in fig.5.1, the branches of the tree traversed in the correct execution path for that program would only be the following ones:

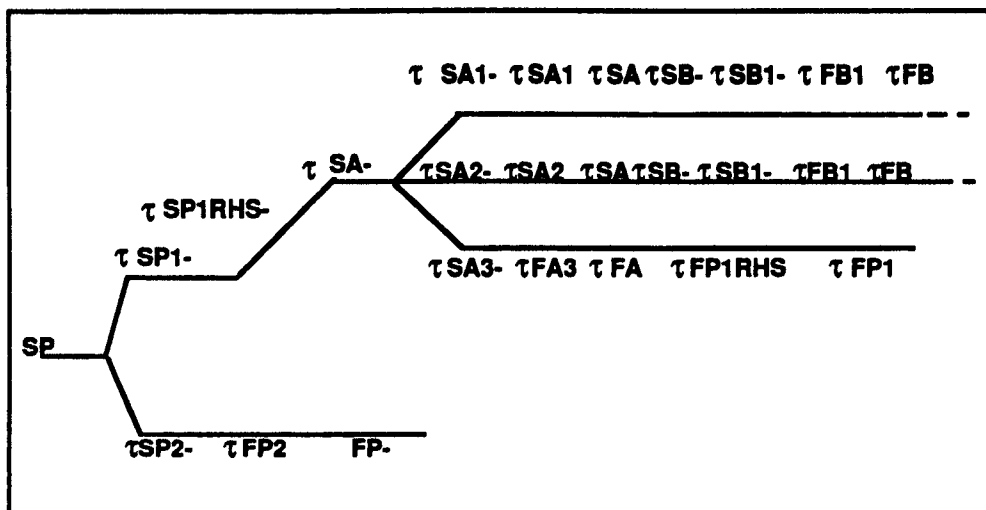


Fig.5.2 Section of semantic tree showing branches traversed by the Prolog interpreter in normal search

If however looking at the diagram shown overleaf in fig.5.3, we determine that in a student's model of backtracking only the branches indicated by darker lines had been traversed, we begin to have an understanding of that student's model of the interpreter. The path follows the first line of successful communications, leading on to the

unsuccessful communication "τ FB1" and subsequently to "FP-", signifying the failure of the program. This indicates that the student suffers from the misconception that on the first failure of a subgoal, the program fails, i.e. the interpreter does not backtrack. As we discussed in chapter two, this is a backtracking model which has been previously documented [Coombs & Stell 1985], [Taylor 1987], [Fung 1987 b] and is usually referred to as the 'try once and fail' or 'try once and pass' model.

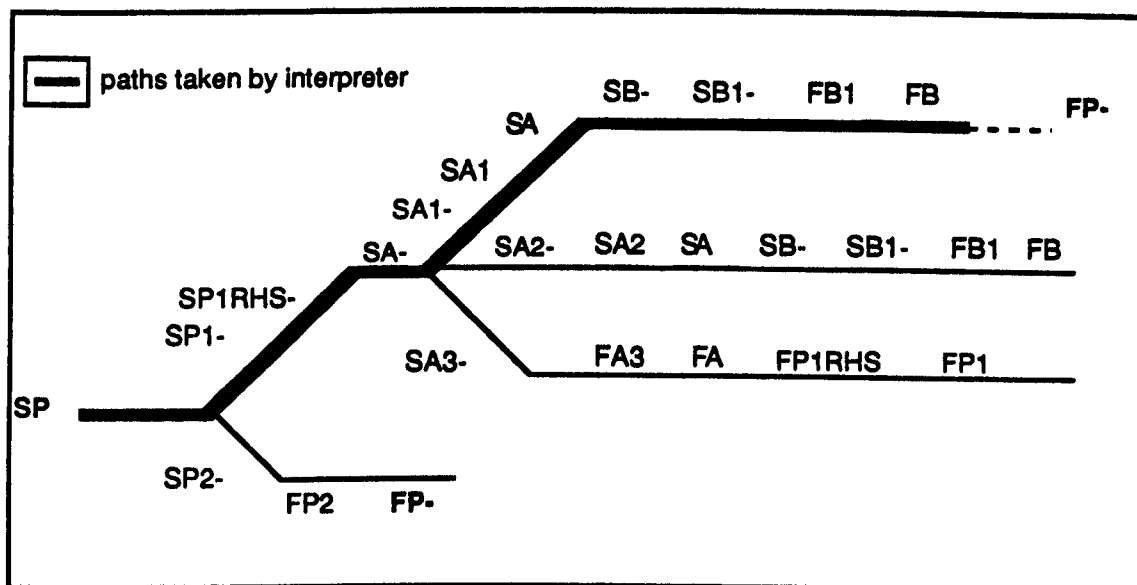


Fig.5.3 Branches of CCS tree traversed in 'try once and pass' model of interpreter

The process of determining which model of the interpreter a student is exhibiting and what this tells us about a student's understanding of the backtracking process is basically one of search to reconstruct the path taken through the semantic representation tree, as reflected by student input. The approach adopted for this reconstruction has been to incorporate the process within the framework of a production rule interpreter.

Such an approach has been taken because it provides two advantages, the first of which is discussed in this section, the second of which is postponed for discussion until [section 5.4].

This first advantage lies in the possibilities it offers of dynamically constructing a set of rules from student input, which would reflect the particular model of the interpreter held by that student. As we indicated earlier, a serious disadvantage of the 'bug library' approach to interpreting student input in tutoring systems is the inability of the system to interpret any student input other than that which corresponds to stored system data. The system being developed here can generate the more common 'known' misconceptions of the Prolog interpreter for a given program, rather than having to prestore them. This is an important step in the development of this approach to student modelling and is in itself a move away from the concept of prestoring the likely 'bugs' for each program. The ultimate goal of further research however, would be to have the flexibility to reconstruct and generate the actual patterns of misconceptions found in students' models of backtracking, in cases where these do not conform to the more usual or expected patterns.

This reconstruction would require a clean way of unambiguously describing execution as represented by the directions taken through the semantic representation. A production rule technique therefore seems best suited to meet that need and worth adopting in view of this long term potential. Using such a technique allows us to reduce the process of execution, i.e. the path through the search tree, into terms of specific situations in which specific actions are taken, a condition/action cycle reflected in the rules of the system. In the following sections we look at the process of developing the primitives of this system

5.2 Developing rule conditions

Given the starting position of execution on the semantic tree, the production rule interpreter cycles through a set of rules, attempting to match the left hand side of a rule with the data provided about that position. On finding a match, the right hand side of that rule fires, producing a new position on the tree. This is added to working memory and the process repeated using this new position.

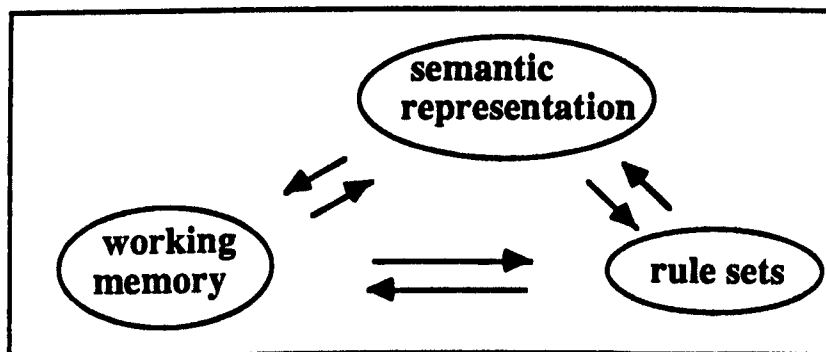


Fig.5.4 Components of production rule interpreter

The cycle continues until a position is encountered which, when matched, fires the stopping condition. At this point the system returns the list of positions which has been built up in working memory and which represents the execution path taken through the semantic tree. Given a clean description of situations and actions, recording the actions taken by a student in specific situations should allow the system to construct a rule set which will generate that student's model of execution. Producing this description language was the first task in developing the production system. In principle, whether 'correct' or 'faulty', the execution path can be described informally in terms of conditions and actions as follows:

**((if at a given position, going in a particular direction and given a certain situation)
(take a certain action))**

In the following section we look more closely at the left hand side of the rules we have developed, which take the following syntactic form:

((position ?node)(direction y)(type ?node z)

where '?node' will instantiate to a node position, 'y' to the value of either 'backward' or 'forward' and 'z' to the classification of the node concerned. These elements will each be discussed in the following sections. The (direction) element when instantiated to forward, indicates that the execution path is moving forward in direction and conversely when instantiated to backward that the execution path is moving in a backward direction. The element (type ?node) differentiates between the type of nodes which in turn effect the element (position ?node) in relation to actions which take place when the rules are instantiated.

5.2.1. Node-types

We look first at the left hand side element (type ?node). This section discusses why there is a need to differentiate between the types of nodes and looks at the way in which this has been done. For the purposes of illustrating the points discussed we refer back to the semantic representation of the program p if a & b shown in fig.5.1.

(1) Disjunct nodes

If we were to follow an execution path through this CCS representation we would start from the beginning of the tree at the node 'SP' (fig.5.1). This node leads to either 'τ SP1-' or 'τ SP2-'. At this point a choice must be made, either to follow the 'τ SP1-' branch or to follow the 'τ SP2-' branch (the relevant section is shown overleaf). If we were to choose to follow the 'τ SP1-' branch along, via the next node 'τ SP1RHS-' we

could then move to the node ' τ SA-'. Here there is again a choice to be made, this time to follow either the branch ' τ SA1-' the branch ' τ SA2-' or the branch ' τ SA3-' (shown below):

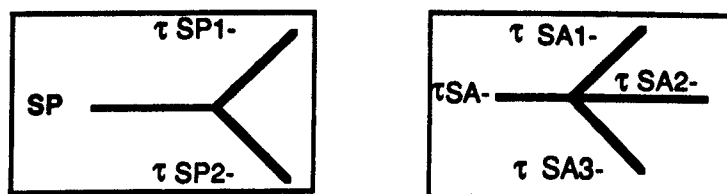


Fig.5.5 Disjunct points with same names

In each case the nodes ' τ SP' and ' τ SA-', are characterised by the fact that at that position there is a choice to be made as to which branch to follow next. Both nodes can be similarly described as 'disjunct' nodes, since they are both nodes at which we have a choice of paths to follow, in the first case either ' τ SP1-' or ' τ SP2-', in the second ' τ SA1-', ' τ SA2-' or ' τ SA3-'.

(2) Disjunct nodes

Another situation however, arises at the node position ' τ SP1RHS-' (fig.5.1), at which point in the above section we moved to ' τ SA-'. The node ' τ SP1RHS-' is also a disjunct, since at this point there is a choice between node ' τ SA-' and the node ' τ SB-'. It represents a different type of choice, however. Whereas the choices shown in fig.5.5 were between nodes of the same name, i.e. ' τ SP1-' and ' τ SP2-' or ' τ SA1-', ' τ SA2-' and ' τ SA3-', the choice here is between nodes with different names, i.e. ' τ SA-' and ' τ SB-'. The relevant section is shown below (fig.5.6).

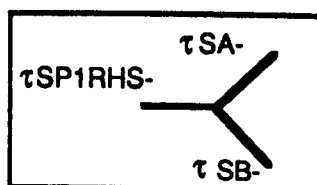


Fig.5.6 Disjunct point with different names

Behaviour, in terms of actions taken at these two sorts of disjunct would not necessarily be the same, therefore a differentiation must be made between them. This has been done in the naming process, so while the nodes referred to above in fig.5.5, 'SP' and 'τ SA-' are described as (type ?node disjunct), the node referred to in fig.5.6, 'τ SP1RHS-' is referred to as (type ?node disjunctrhs). The final letters 'rhs' signal that the branches following it contain the machines of the right hand side of that clause, in this case the 'τ SP1-' clause. In the program, 'p if a & b', which the CCS representation of fig.5.1 describes, the 'right hand side' of clause 'p', is 'a & b'. Since the CCS representation allows for all possible execution paths through the program, at the point of contacting the right hand side of clause 'p', i.e. at the node 'τ SP1RHS-', either subgoal, 'a' or 'b', could be pursued first. In terms of following an execution path through the semantic representation of fig.5.1, this means that at the node 'τ SP1RHS-' either the branch beginning with the node 'τ SA-' or the branch beginning with the node 'τ SB-' could be followed. In using the semantic representation to model normal Prolog search however, the execution path at such a disjunct would always move to the first branch leading from the disjunctrhs i.e. to the node 'τ SA-' rather than the node 'τ SB-'. This reflects the search order followed by the Prolog interpreter, which orders the subgoals of a clause in strict left to right direction, trying to prove subgoal 'a' before subgoal 'b'. The subgoal 'b' would never be investigated before the subgoal 'a' and if this investigation failed, the subgoal 'b' would never be tried. In terms of following the execution path using the semantic representation, this means that in normal Prolog search the branch beginning with the node 'τ SB-' would never be followed. The movement of the execution path at a disjunctrhs, if we are modelling normal Prolog search then, is always to move to the

node on the first branch leading from the disjunctrths. It is necessary to distinguish these 'disjunctrths' nodes from other disjuncts, since in constructing an execution path from student input, the system must be able to recognise the situation when a student does not appreciate the importance of the Prolog goal ordering process.

(3) Disjunctprime nodes

Another case where a 'different name disjunct' occurs in the CCS tree is in the semantic representation of the 'and' where there are more than two subgoals to be proved. This is due to the parallel interpretation of the conjunction of goals of the right hand side, which offers a non-deterministic choice of execution paths. The type of disjunct in this case is classed as (type ?node disjunctprime). An example of this is shown below (fig.5.7) in a section of a CCS tree in which there are three right hand elements: 'p if a & b & c.' The type of disjunct referred to is illustrated below by the node ' τ SA', representing the successful communication of ' τ SA-' via ' τ SA1', and then offering a choice of communication paths, ' τ SB-' or ' τ SC-' (outlined in the top right hand corner of the diagram below, fig.5.7).

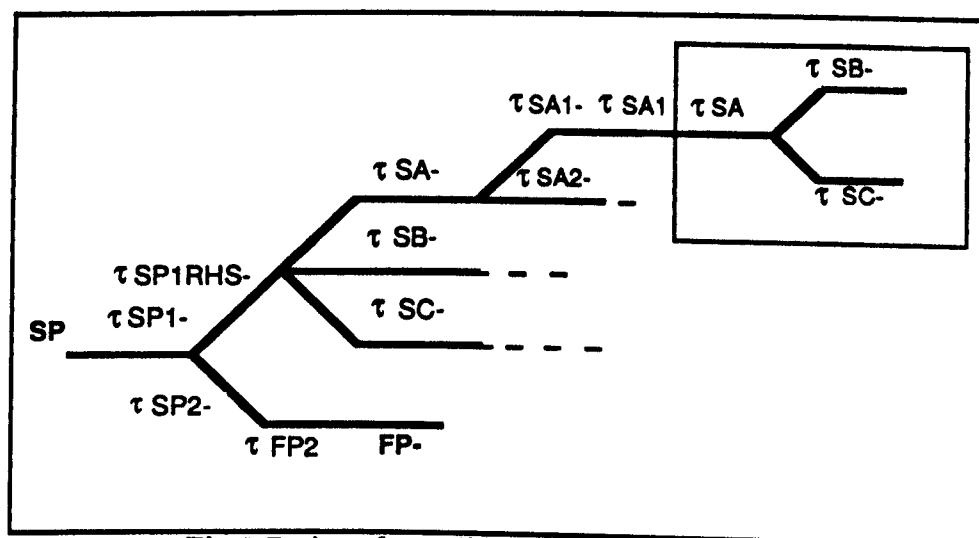


Fig.5.7 Another 'different name' disjunct

Again, due to the strict left to right ordering of the subgoals by the Prolog interpreter, at the disjunct ' τ SA' in fig.5.7, the subgoals will always be satisfied in the order, 'a', then 'b' then 'c' as the interpreter scans from left to right. If the execution path is at the point ' τ SA' in the highlighted area in the fig.5.7, which can be interpreted as at the point of having satisfied the subgoal 'a', then the next subgoal to be satisfied is 'b' (represented here by the communication path ' τ SB-'), in normal search it would never be 'c' (' τ SC-') at this stage. This type of disjunct, found in the parallel interpretation of the 'and' component, is referred to as a 'disjunctprime'. In normal Prolog search, only the first of the branches offered, in this case ' τ SB-', is traversed, either in a forward or backward direction. The execution path followed at this type of node is, as with the 'disjunctrhs' type of node, simply a move to the node of the first branch leading from this disjunct. However these nodes too, must also be distinguished as a particular type of disjunct, in order for the system to be able to identify abnormal choices of execution paths. This is essential for the task of modelling and generating faulty or incomplete models of the Prolog interpreter.

(4) Nochoice nodes

The 'nochoice' type of nodes referred to above are those such as the nodes highlighted in the diagrams overleaf in fig.5.8. As opposed to the types of disjunct nodes discussed above, they do not involve a choice. Progress through the search space, or communication between such nodes, basically consists of moving to the next position along the branch or back to the previous position on the branch. These nodes are described as (type ?node nochoice-nodes).

systematically numbering the nodes of the given semantic representation, each node is allocated a unique position according to its level in the semantic tree. In the tree of fig.5.1, for example, the position of the initial node 'SP' would be identified as (position (1)). At the next level, 'τ SP1-' would be (position(1 1)) and 'τ SP2-' would be (position (1 2)). Numbered in this way, each node has its own position identifier which the system uses in conjunction with the node-type information in the process of instantiating the left hand sides of rules. Putting the three elements, position, direction and type, together, the left hand side of a rule such as the following, for example:

((position ?node)(direction forward)(type ?node disjunct))

could be instantiated by 'SP' of fig.5.1, to:

((position (1))(direction forward)(type (1)disjunct))

but not by 'τ SP1-', since in this case the node type would not match, 'τ SP1' being a (type nochoice-node). To take another example:

((position ?node)(direction forward)(type ?node nochoice-node))

could be instantiated by 'τ SP1-' to:

((position (1 1)) (direction forward)(type (1 1) nochoice-node)),

or by 'τ SA1-' to:

((position (1 1 1 1 1))(direction forward)(type (1 1 1 1 1) nochoice-node))

Information about the position and type of nodes is computed from the CCS tree and given as input to the production rule system by a function initialising working memory [appendix B3]. This takes the semantic tree in list form as input and produces as output a list in which each node is allocated a number and a type. A call to the function 'initwm' with the semantic representation of the program *p* if *a* from fig.4.14

[section 4.4] for example would initialise working memory with the following data:

```
((POSITION (1)) (DIRECTION FORWARD))
((CHOICEPOINT (1)) (TYPE (1) SAME-NAME-DISJUNCT) (NAME (1) SP))
((CHOICEPOINT (1 1)) (TYPE (1 1) NOCHOICE-NODE)(NAME (1 1) "τSP1-"))
((CHOICEPOINT (1 1 1)) (TYPE (1 1 1) NOCHOICE-NODE)(NAME (1 1 1) "τSP1RHS-"))
((CHOICEPOINT (1 1 1 1)) (TYPE (1 1 1 1) SAME-NAME-DISJUNCT)
(NAME (1 1 1 1) "τSA-"))
((CHOICEPOINT (1 1 1 1 1)) (TYPE (1 1 1 1 1) NOCHOICE-NODE)
(NAME (1 1 1 1 1) "τSA1-"))
((CHOICEPOINT (1 1 1 1 1 1)) (TYPE (1 1 1 1 1 1) NOCHOICE-NODE)
(NAME (1 1 1 1 1 1) "τSA1"))
((CHOICEPOINT (1 1 1 1 1 1 1)) (TYPE (1 1 1 1 1 1 1) NOCHOICE-NODE)
(NAME (1 1 1 1 1 1 1) "τSA"))
((CHOICEPOINT (1 1 1 1 1 1 1 1)) (TYPE (1 1 1 1 1 1 1 1) NOCHOICE-NODE)
(NAME (1 1 1 1 1 1 1 1) "τSP1RHS"))
((CHOICEPOINT (1 1 1 1 1 1 1 1 1)) (TYPE (1 1 1 1 1 1 1 1 1) NOCHOICE-NODE)
(NAME (1 1 1 1 1 1 1 1 1) "τSP1"))
((CHOICEPOINT (1 1 1 1 1 1 1 1 1 1)) (TYPE (1 1 1 1 1 1 1 1 1 1) FINAL-SUCCESS)
(NAME (1 1 1 1 1 1 1 1 1 1) SP-))
((CHOICEPOINT (1 1 1 1 2)) (TYPE (1 1 1 1 2) NOCHOICE-NODE)
(NAME (1 1 1 1 2) "τSA2-"))
((CHOICEPOINT (1 1 1 1 2 1)) (TYPE (1 1 1 1 2 1) FAILCHOICE)
(NAME (1 1 1 1 2 1) "τFA2"))
((CHOICEPOINT (1 2)) (TYPE (1 2) NOCHOICE-NODE)
(NAME (1 2) "τSP2-"))
((CHOICEPOINT (1 2 1)) (TYPE (1 2 1) FAILCHOICE)
(NAME (1 2 1) "τFP2"))
```

5.3. Rule actions

The data which is obtained from the semantic tree and used to initialise working memory is used by the production rules in the instantiation of the left hand sides of rules. Each node of the semantic representation is identified uniquely by the initialisation process. Thus at any one state of the process the data extracted from working memory can successfully instantiate the left hand side of one rule only. The system does not therefore incorporate any mechanism for conflict resolution.

The initial element of the list above, ((position(1))(direction forward)) is the starting point in working memory, representing the first step in the execution path. We now go on to look at the form of the right hand sides, the actions to be taken when the left hand side conditions are

met. Again, we are interested in describing the actions which can be taken in as clear a way as possible with the objective of using their descriptions as building blocks in constructing an execution path. For the purposes of discussing this, we will take as example the actions, i.e. right hand sides, of rules from a ruleset designed to model the normal execution path of a Prolog program. The actions which can be taken are of two sorts, (i) those of moving forward or backward to the next position on a particular branch and (ii) those of moving up or down from one branch of the tree to a higher or lower numbered one. For example in the diagram of fig.5.9 below, if we move from ' τ SA1-' to ' τ SA1', this is a forward movement along a branch a level into the tree, while to move back from ' τ SA1' to ' τ SA1-' would be a backward movement along the same branch. However, to move from ' τ SA1-' to ' τ SA2-' is to move up to a higher branch, while to move from ' τ SA2-' to ' τ SA1-' would be to move down to a lower branch.

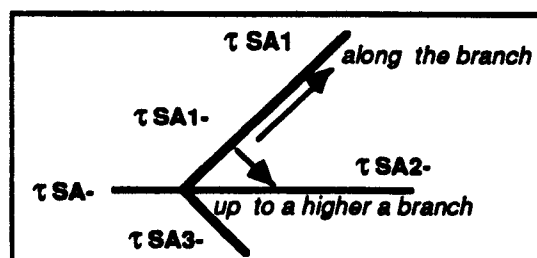


Fig.5.9 Directions of actions

These two different types of movement are represented in the system as 'next-level' and 'up-levels' actions. Which kind of action is taken on the firing of a right hand side of a rule is determined by the type of node specified in the left hand side.

5.3.1. Action at nochoice-nodes

In the course of normal Prolog execution, 'nochoice-nodes' do not offer any choice of action. Where for instance, the left hand side of a rule has

been successfully instantiated to a 'nochoice-node' the action is 'next-level' and the position at one move along the branch concerned is added to working memory. When the node 'τ SA1-' in fig.5.9 instantiates the following rule:

((position ?node)(direction forward)(type ?node nochoice-node))
 (left hand side)
((position (next-level ?node))(direction forward))
 (right hand side)

to:

((position (11111)) (direction forward)(type (11111) nochoice-node))
((position (next-level (11111))) (direction forward))

the right hand side fires and the function 'next-level', given the node position (11111) as input computes the node position (11111), and the next step in the execution path added to working memory is:

((position (11111))(direction forward))

indicating in this case that the execution path moves on to 'τ SA1'. The equivalent action where the execution path is moving in a backward direction after the failure of a node to communicate successfully, is to step back to the previous node. When, for instance, node 'τ SA1-' instantiates the rule:

((position ?node)(direction backward)(type ?node nochoice-node))
((position (stepback ?node))(direction backward))

to:

((position (11111)) (direction backward)(type (11111) nochoice-node))
((position(stepback (11111))) (direction backward))

the right hand side fires and the function 'stepback', using (position (11111)) as input, outputs (position (1111)) and the new position added to working memory is:

((position (1111))(direction backward))

indicating that the execution path has moved back from 'τ SA1-' to 'τ SA-'.

5.3.2. Action at disjunctrhs and disjunctprime nodes

In normal Prolog execution, action taken at a 'different name' disjunct, that is, either a (disjunctrhs) or a (disjunctprime), would be similar to that taken at a 'nochoice-node', moving to the next position along the tree in a forward direction, e.g.

```
((position ?node)(direction forward)(type ?node disjunctrhs))
((position (next-level ?node ))(direction forward)))
```

When the execution path is moving in a forward direction, the node 'τ SP1RHS-' of fig.5.1 would, for instance, instantiate the above rule to:

```
((position (111))(direction forward)(type (111) disjunctrhs))
((position (next-level (111) ))(direction forward)))
```

and the new position added to working memory would then be:

```
((position (1111) ))(direction forward)))
```

On reaching an unsuccessful communication the action for both the disjunctrhs and the disjunctprime node would be like that taken at a 'nochoice' node and the new position added to working memory would be a 'stepback' along the branch.

5.3.3. Actions at disjunct nodes

The situation at a (type ?node disjunct) is different however, as discussed earlier in section 5.2.1 (1) and 5.2.1 (2). It can be seen if we use fig.5.5 as an illustration, that the action taken at this type of node would vary according to the stage of execution. We shall look at the action taken where the execution path arrives at a node in a forward direction

first, then consider the two cases which can arise when the execution path is moving in a backward direction, i.e. the program is backtracking.

(1) In a forward direction

In normal Prolog search, if the node has been reached for the first time as the execution path is moving in a forward direction, then the action taken would be simply to go on to the next level, which would be the first branch of that choice. This is captured in the rule:

```
((position ?node)(direction forward)(type ?node disjunct))
((position (next-level ?node ))(direction forward)))
```

In fig.5.5 for instance, at the node 'SP' this action would lead to 'τ SP1-', or at the node 'τ SA-' would lead to 'τ SA1-'.

(2) In a backward direction

If the execution is moving in a backward direction, as would be the case after an unsuccessful communication, the execution path would eventually lead back to a disjunct node. In this case, reaching a disjunct node in a backward direction, the action taken would normally be one of going up a level to try a higher branch. In terms of Prolog search, this would be looking for other ways of satisfying the clause. The execution path in this instance would then go up from 'SP' to 'τ SP2-' and in the case of 'τ SA-', to 'τ SA2-' rather than, respectively, to 'τ SP1-' or to 'τ SA1-', so the right hand side of the rule for such a case should be:

```
((position ?node)(direction backward)(type ?node disjunct))
((position (go up a level ?node ) )(direction forward)))
```

However, this process could be repeated, since the next branch of the execution path could also fail. This would mean that the disjunct point

was again approached in a backward direction and the same rule would fire. When the point has been reached at which there are no more successful communication paths to follow from that disjunct, the appropriate action is then one of stepping back along the tree. If, for instance, all the branches leading from the disjunct 'τ SA-' in fig.5.5 had been followed and had led to failed communications, then the execution path would be to step backwards from 'τ SA-'. There are therefore two different actions which might be appropriate on arriving at a disjunct when the execution path is moving in a backward direction. This must be reflected in two rules which allow for the two differing sets of circumstances. One must allow for the situation in which at a return to a disjunct, there are higher untried branches, in which case the action is to move up a level to try a higher branch, the other must allow for the situation in which there are no more untried higher branches, in which case the action is that of stepping back. In order to identify these circumstances, the system needs the ability to distinguish between subsequent returns of the execution path to such a disjunct. It must have some means of knowing at each return whether or not there are higher level branches and if so, to know which are still untried. A constraint, (up-levels (next-level ?node)) designed to provide this information has been incorporated in the left hand side of each of the two rules concerned, which for convenience are discussed below as disjunct rule (a) and disjunct rule (b).

(disjunct rule a)

**((position ?node)(direction backward)(type ?node disjunct)
 (up-levels (next-level ?node))))
 ((position (up-levels (next-level ?node)))(direction forward)))**

The constraint takes the form of a function which uses information in working memory for two purposes. One is to check on positions

previously recorded and the other is to check that there are higher branches at the given disjunct. The function 'up-levels' uses as input the lowest level branch leading from the disjunct in question, i.e. (next-level ?node). At disjunct 'τ SA-' of fig.5.5 for example this would be 'τ SA1-'. It searches the positions recorded in working memory to check whether or not a branch, one level up, leading from the disjunct has previously been recorded. In fig.5.5 this would be 'τ SA2-' (the first level branch leading from the disjunct will, in normal Prolog search, always have been used earlier as the execution path moved forward). If no second level position is found to have been previously recorded, the function is successful, i.e. the constraint has been satisfied and the action of rule (a) is fired, adding the position of this second level branch to working memory. If however the second level branch leading from the disjunct had been recorded in working memory, indicating that it had been traversed, the function would repeat the search process at the next highest level. In fig.5.5 this would mean checking the list of recorded positions for 'τ SA3-'. If this too had been used, the search process would continue, a level higher at each iteration. At each level which is searched, the data in working memory is also checked to confirm that there is such a branch in the CCS tree. In the case of the disjunct in fig.5.5, this would reveal that there was no branch beginning with 'τ SA4-', so the function would return a nil result. In this case, rule (a) would fail to be instantiated and the second rule relevant to the situation would be applied.

(disjunct rule b)

```
(((position ?node)(direction backward)(type ?node disjunct)
  ((not(up-levels (next-level ?node))))))
((position (stepback ?node))(direction backward)))
```

In the case of rule (b), the constraint (up-levels(next-level ?node)) of rule (a) is negated. The function 'up-levels' must return a nil result in order to satisfy the constraint of rule (b), i.e. it must show that there are no untried higher level branches. As explained above, the function will return a nil result if every branch leading from the disjunct in question has been previously recorded and there are no more higher branches to traverse. In the example of fig.5.5, where all three branches of the disjunct 'τ SA-' in fig.5.5 have been used and no higher branch is left to try, the function 'up-levels' would return the result nil. This would satisfy the negation, so the constraint of rule (b), that there are no higher levels to try, would be satisfied and the right hand side of the rule would fire. The next position back along the tree from the disjunct (stepback ?node) would be added to working memory.

5.3.4. Action at failchoice-nodes

In modelling the normal course of Prolog execution, the action taken at a 'failchoice-node' is always one of stepping back to the previous node.

**((position ?node) (direction forward)(type ?node failchoice-node))
((position(stepback ?node)) (direction backward))**

The important effect of this rule firing and adding a new position to working memory is that of changing the direction of the execution path being constructed. From that point only the rules relating to nodes being approached in a backward direction will be applicable, unless a disjunct is encountered which instantiates a rule resetting the execution path to a forward direction.

5.3.5. Action at final nodes

There are two situations in which the action of a rule should be to halt the system. One is when the left hand side of a rule instantiates to a successful completion of the program.

```
((position ?node)(direction forward)(type ?node final-success))
(halt))
```

This happens when the position reached relates to the successful complementary node of the start node. In the case of the program 'p if a', this would be 'SP-', denoting a successful communication path from 'SP'. In the case of our breakfast program it would be 'SBREAKFAST-', denoting a successful communication path from 'SBREAKFAST'.

The other situation arises when the execution path has traversed all the branches available in the course of normal Prolog search and has not been able to establish a successful communication path. At this point the program finally fails and the action is to add halt to working memory.

```
((position (1))(direction backward)(type ?node disjunct)
(used(up-levels(next-level ?node))))
(halt))
```

In this case the instantiation of the left hand side relates to the starting position of the program, but arriving at it in a backward direction. At the beginning of the program 'p if a' for example, working memory is initialised to ((position(1))(direction forward)). When the most recent addition to working memory is ((position(1))(direction backward)) and given that the constraint (not(up-levels)) is satisfied, i.e. that there are no higher untried branches, then the program finally fails since the last branch has ended in 'FP-', denoting an unsuccessful communication.

5.4. A ruleset for normal Prolog search

Putting together the left and right hand side of the rules we have discussed gives us then the following ruleset for Prolog search:

```

(((position ?node)(direction forward)(type ?node nochoice-node)
  ((position (next-level ?node ))(direction forward)))

(((position ?node)(direction backward)(type ?node nochoice-node)
  ((position (stepback ?node ))(direction backward)))

(((position ?node)(direction forward)(type ?node disjunctrhs)
  ((position (next-level ?node))(direction forward)))

(((position ?node)(direction backward)(type ?node disjunctrhs)
  ((position (stepback ?node))(direction backward)))

(((position ?node)(direction forward)(type ?node disjunct)
  ((position (next-level ?node))(direction forward)))

(((position (1))(direction backward)(type (1) disjunct)
  ((not(up-levels(next-level (1))))))
  ((halt)))

(((position ?node)(direction backward)(type ?node disjunct)
  ((up-levels (next-level ?node))))
  ((position (up-levels(next-level ?node)))(direction forward)))

(((position ?node)(direction backward)(type ?node disjunct)
  ((not(up-levels (next-level ?node))))))
  ((position(stepback ?node))(direction backward)))

(((position ?node)(direction forward)(type ?node disjunctprime)
  ((position (next-level ?node ))(direction forward)))

(((position ?node)(direction backward)(type ?node disjunctprime)
  ((position (stepback ?node ))(direction backward)))

(((position ?node)(direction forward)(type ?node failchoice)
  ((position (stepback ?node))(direction backward)))

(((position ?node)(direction forward)(type ?node final-success)
  ((halt))))

```

Fig.5.10 Ruleset for normal Prolog search

As described earlier, given an initial starting point ((position(1)) (direction forward)), the system begins a cycle through these rules in a search to instantiate a left hand side of a rule with this position. When this is successful, the right hand side of the matching rule is fired, adding a new position to working memory. On the next cycle this new addition to working memory is used to instantiate the left hand side of another rule, causing its right hand side to fire, resulting in another position being added to working memory. This search process continues, the most recent position added to working memory being used on each cycle, until a rule is met, the right hand side of which adds ((halt)) to working memory. At this point the system returns the execution path of the program constructed from the list of positions in working memory. A call to the function 'beginsearch' to model the execution path of proving the program:

p if a.
a.

for example, would cycle through the ruleset building the following list of positions in working memory:

((POSITION (1))
(POSITION (11))
(POSITION (111))
(POSITION (1111))
(POSITION (11111))
(POSITION (111111))
(POSITION (1111111))
(POSITION (11111111))
(POSITION (111111111))
(POSITION (1111111111)))

stopping when the position relating to 'SP-' instantiates the rule whose action adds ((halt)) to working memory. These positions are related to

the relevant nodes of the semantic representation and the system outputs the successful execution path:

```
(SP
  "τ SP1-"
  "τ SP1RHS-"
  "τ SA-"
  "τ SA1-"
  "τ SA1"
  "τ SA"
  "τ SP1RHS"
  "τ SP1"
  SP-)
```

Similarly a call to prove 'p' in the program:

```
p if a & b.
a.
a.
```

builds a list of positions, relates these to the relevant nodes and outputs the following execution path which in this case however, reflects the backtracking process that has taken place in the course of the program failing. In this and subsequent examples of output the columns are to be read top to bottom, left to right. In the sample output below, in order to make it easier for the reader to follow, the points at which backtracking occurs have been marked by a double asterisk.

(SP	"τ SB1-"	"τ SB-"	"τ FA"
"τ SP1-"	"τ FB1"***	"τ SB1-"	"τ FP1RHS"
"τ SP1RHS-"	"τ FB"	"τ FB1"***	"τ FP1"
"τ SA-"	"τ SA-"	"τ FB"	FP-
"τ SA1-"	"τ SA2-"	"τ SA-"	"τ SP2-"
"τ SA1"	"τ SA2"	"τ SA3-"	"τ FP2"
"τ SA"	"τ SA"	"τ FA3"***	FP-)
"τ SB-"			

Having built our rule set for Prolog normal search, we are now in a

position to discuss the second advantage of using a production rule approach. This is the modularity inherent in such an approach which makes it a relatively easy matter to add or remove rules from the basic set in order to model the more common misconceptions of the Prolog interpreter. In the following section we look at how this modularity allows us to adapt the ruleset shown in fig.5.10 in order to undertake such modelling.

5.5. Modelling misconceptions - some examples

In this section we discuss the task of modelling novices' misconceptions of the Prolog interpreter. Within the framework of a production rule interpreter, this task is primarily one of expressing the behaviour we wish to describe by adding rules, or omitting rules from the ruleset describing normal Prolog search. Examples illustrating this are given below.

5.5.1. Try once and pass

As discussed earlier, the faulty model of the interpreter in this misconception is that on the failure of a subgoal the program fails without any attempt to backtrack and resatisfy earlier goals. This model can be generated simply by replacing the rule which applies to behaviour at a 'failchoice' node. In the correct model of the Prolog interpreter, when the execution path arrives at an unsuccessful communication, i.e. a 'failchoice' node, the action is to 'stepback' one position to initiate backtracking:

```
((position ?node)(direction forward)(type ?node failchoice))  
((position (stepback ?node))(direction backward)))
```

To model the 'try once and pass' behaviour, this is replaced by the rule:

```
(((position ?node)(direction forward)(type ?node failchoice))
  ((halt)))
```

Consequently, on reaching a position which satisfies a failchoice-node, the right hand side of the rule fires and 'halt' is added to working memory. The normal execution path shown above in [section 5.4] for the program:

```
p if a & b.
a.
a.
```

would be replaced by the following execution path, ending prematurely without showing any backtracking, so reflecting the try once and pass misconception.

```
(SP          "τ SA1-"          "τ SB1-"          "τ FP1"
  "τ SP1-"    "τ SA1"          "τ FB1"          FP-)
  "τ SP1RHS-" "τ SA"          "τ FB"
  "τ SA-"     "τ SB-"          "τ FP1RHS"
```

5.5.2. Redo from left

Another misconception 'redo from the left' discussed earlier is one in which the student believes that on failing a subgoal the execution path returns to the leftmost subgoal of the clause before trying to resatisfy previously successful subgoals. Modelling this within the production rule framework can be achieved by replacing the two disjunctrhs rules and simplifying the disjunct rules which are normally applicable when the execution path is moving in a backward direction.

In normal backtracking when the execution path returns to a disjunct in the course of the stepback process, any untried higher branches will be followed. In this faulty model of backtracking the execution path simply steps back through any such disjuncts between the failed subgoal and the leftmost subgoal, so the disjunct rule for backtracking:

```
((position ?node)(direction backward)(type ?node disjunct)
  ((up-levels (next-level ?node))))
  ((position (up-levels(next-level ?node)))(direction forward)))
```

is omitted to reflect this, and the second disjunct rule for backtracking:

```
((position ?node)(direction backward)(type ?node disjunct)
  ((not(up-levels (next-level ?node))))
  ((position(stepback ?node))(direction backward)))
```

is simplified to:

```
((position ?node)(direction backward)(type ?node disjunct)
  ((position(stepback ?node))(direction backward)))
```

since there is no need to check for higher branches before stepping back. This results in the stepping back process continuing until a disjunctrhs is encountered. As mentioned earlier, the distinguishing feature of a disjunctrhs node is that it marks the beginning of the right hand side of a clause. In the redo from left model the execution path returns to this point before beginning a search for higher branches to resatisfy the subgoals of the clause. In effect the action then taken at this node is based on the same criteria as those taken into account at a disjunct node in the normal Prolog backtracking model. The disjunctrhs rule in a backtracking situation is therefore not adequate in its original form:

```
((position ?node)(direction backward)(type ?node disjunctrhs)
  ((position (stepback ?node))(direction backward)))
```

Before stepping back it must now incorporate the constraint that there are no higher levels of previously tried branches to follow:

```
(((position ?node)(direction backward)(type ?node disjunctrhs)
  ((not(up-levels (following ?node))))))
  ((position (stepback ?node))(direction backward)))
```

If this constraint is not met, thus indicating that there are higher levels of previously tried branches to follow, then this rule will fail to be instantiated, so as with the disjunct node in normal backtracking, a second disjunctrhs rule is needed to cover this case:

```
(((position ?node)(direction backward)(type ?node disjunctrhs)
  ((up-levels (following ?node))))))
  ((position (up-levels (following ?node)))(direction forward)))
```

The difference between these disjunctrhs rules and those for a disjunct in normal Prolog backtracking is that at a disjunctrhs node it is not the (next-level ?node) which is being tested for higher levels of previously tried branches, but the (following ?node). The latter is the equivalent of the (next-level(next-level ?node)). An illustration will probably make it clear why this difference exists. If we take the case of backtracking in the program:

```
p if a & b & c.
a.
b.
b.
```

at the failure of subgoal 'c', represented in the CCS tree execution path by the node 'τ FC1', then in normal backtracking, flow of execution would stepback until the disjunct 'τ SB-' was encountered. A search for any higher levels of branches leading from this disjunct would then be made in the process of trying to resatisfy the subgoal 'b', such as a branch beginning with 'τ SB2-'. However, at the failure of subgoal 'c' in the redo from left model, the subgoal 'b' would be ignored and this is mirrored in the CCS representation in that the execution path would

ignore the disjunct ' τ SB-' and continue to move back to the disjunctrths position of node ' τ SP1RHS-', shown in the diagram overleaf (fig.5.11):

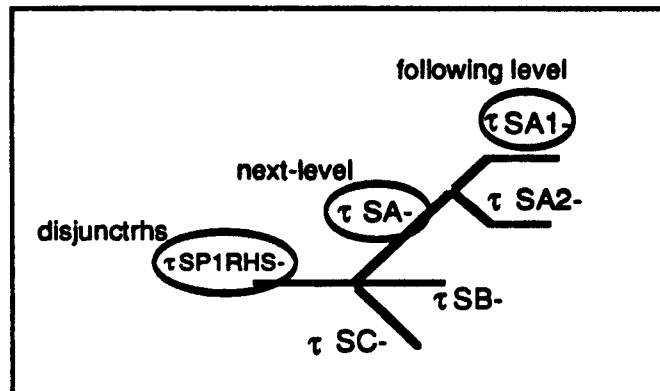


Fig.5.11 Returning to disjunctrths

At this point, the branches leading from the node ' τ SP1RHS-' are differently named. A move to follow a higher level branch would lead to the branch ' τ SB-' being followed, whereas the execution path at that point would be expected to follow a higher level of the branch ' τ SA-', i.e. ' τ SA2-' or ' τ SA3-'. To model this behaviour, the function 'up-levels' takes as input not the 'next-level' position, but the next but one position. In this example for instance, it is the position of node ' τ SA1-' which is needed as input to the function 'up-levels' i.e. not the position output from the function 'next-level', but rather the position which is output by computing the 'next-level (next-level)' node.

Having replaced the original disjunctrths rules for backtracking cases, omitted one of the original disjunct rules for backtracking and simplified another, we now have a ruleset which will model the redo from left misconception. In contrast to the execution path which would be produced using the normal search ruleset to the CCS representation of the program given above, i.e.

(SP	"τ SC1-"	"τ SB-"	"τ FB"
"τ SP1-"	"τ FC1"	"τ SB1-"	"τ SA-"
"τ SP1RHS-"	"τ FC"	"τ SB1"	"τ SA3-"
"τ SA-"	"τ SB-"	"τ SB"	"τ FA3"
"τ SA1-"	"τ SB2-"	"τ SC-"	"τ FA"
"τ SA1"	"τ FB2"	"τ SC1-"	"τ FP1RHS"
"τ SA"	"τ FB"	"τ FC1"	"τ FP1"
"τ SB-"	"τ SA-"	"τ FC"	FP-
"τ SB1-"	"τ SA2-"	"τ SB-"	"τ SP2-"
"τ SB1"	"τ SA2"	"τ SB2-"	"τ FP2"
"τ SB"	"τ SA"	"τ FB2"	FP-)
"τ SC-"			

a call to the function 'beginsearch' using the 'redo from left' ruleset produces the execution path:

(SP	"τ SB-"	"τ FC1"	"τ FP1RHS"
"τ SP1-"	"τ SB1-"	"τ FC"	"τ FP1"
"τ SP1RHS-"	"τ SB1"	"τ SA-"	FP-
"τ SA-"	"τ SB"	"τ SA2-"	"τ SP2-"
"τ SA1-"	"τ SC-"	"τ FA2"	"τ FP2"
"τ SA1"	"τ SC1-"	"τ FA-"	FP-)
"τ SA"			

5.5.3. Facts before rules

A student having this faulty model of the Prolog interpreter, given the program:

```
p if a & b & c.
a if x.
x.
b.
a.
```

would predict that in order to prove 'p' the execution path would arrive at the fact 'a' before arriving at the rule 'a if x', since the belief is that the interpreter distinguishes between facts and rules and chooses to scan 'facts' before rules which have the same head. To model this with

the production rule interpreter, we need to add a constraint to the rule applicable when the execution path encounters a disjunct while moving in a forward direction. This is in order to check whether or not the branches leading from it relate to rules (signalled by the next but one i.e. a 'following' node being a disjunctrhs) or to facts. In the diagram below, for example, taken from the CCS representation of the above program (given in full in Appendix D):

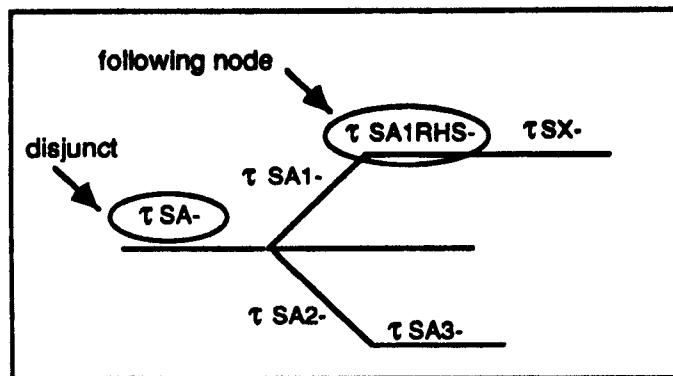


Fig.5.12 Identifying a rule clause

when the execution path is at the disjunct 'τ SA-', in normal search the following rule would apply:

```
((position ?node)(direction forward)(type ?node disjunct)
  ((position (next-level ?node))(direction forward)))
```

This would take the execution path on to the node, 'τ SA1-'. This branch however, leads to the node 'τ SA1RHS-', representing a clause which has a rule as its right hand side. In order to model the behaviour which chooses clauses that are facts, in preference to those which are rules, the system needs to know at the disjunct 'τ SA-' whether or not the node following 'τ SA1-' is a disjunctrhs. This is done by using a constraint in the rule for disjuncts, direction forward:

```
((position ?node)(direction forward)(type ?node disjunct)
  ((rule-clause(following ?node))))
  ((position (up-levels ?node))(direction forward)))
```

The function 'rule-clause' takes as input the results of computing the 'following' node position, which as before, is equivalent to the 'next-level(next-level)' node position. If this position relates to a disjunct, the function 'rule-clause' succeeds and the constraint is satisfied, indicating that the branch leading from the disjunct is the beginning of a rule clause. In this case the action taken is to move up to a higher branch of the disjunct in search of a fact clause. In our example above, the execution path would move from the disjunct 'τ SA-' to the node 'τ SA2-'. If however the function 'ruleclause' fails, indicating that the branch leading from the disjunct is not a rule clause, a second rule for this situation fires:

```
((position ?node)(direction forward)(type ?node disjunct)
  ((not(rule-clause(following ?node))))
  ((position (next-level ?node))(direction forward)))
```

and the execution path proceeds, as in normal search, to the next node.

Modelling this behaviour also necessitates adding a rule to those for approaching disjuncts on backtracking, since there may be untried branches representing rule clauses, which were not traversed as the execution path initially moved forward. If the higher level branches representing fact clauses had all been tried and failure had caused backtracking, then these 'rule' branches, ignored the first time round, would now be tried. In addition therefore to the normal rules covering the action to be taken at disjuncts, the system must include a rule which allows for encountering an untried 'next-level' node during backtracking:

```
((position ?node)(direction backward)(type ?node disjunct)
  ((not(used(next-level ?node))))
  ((position (next-level ?node))(direction forward)))
```

The constraint (not(used(next-level ?node))) checks the list of positions in working memory to see if the 'next-level' node from the disjunct has previously been recorded, if not, then the execution path now follows this branch. In our program above, it would mean that after backtracking through the fact 'a' because the subgoal 'c' had failed, the rule clause 'a if x' would now be tried, i.e. in fig.5.13, the branch beginning with 'τ SA1-' would now be followed. Using these replacement rules for action at a disjunct when the execution path is moving in a forward direction and adding a rule to subsequently catch up on the untried rule clauses when it is moving in a backward direction, gives us a ruleset to model the 'facts before rules' misconception. Applied to the CCS representation of the program above, the resulting execution path is as follows:

(beginsearch aifx rules)

(SP "	"τ FC1"	"τ SA1RHS"	"τ FB2"
"τ SP1-"	"τ FC"	"τ SA1"	"τ FB"
"τ SP1RHS-"	"τ SB-"	"τ SA"	"τ SX-"
"τ SA-"	"τ SB2-"	"τ SB-"	"τ SX2-"
"τ SA2-"	"τ FB2"	"τ SB1-"	"τ FX2"
"τ SA2"	"τ FB"	"τ SB1"	"τ FX"
"τ SA"	"τ SA-"	"τ SB"	"τ FA"
"τ SB-"	"τ SA1-"	"τ SC-"	"τ SA3-"
"τ SB1-"	"τ SA1RHS-"	"τ SC1-"	"τ FA3"
"τ SB1"	"τ SX-"	"τ FC1"	"τ FA"
"τ SB"	"τ SX1-"	"τ FC"	"τ FP1RHS"
"τ SC-"	"τ SX1"	"τ SB-"	"τ FP1"
"τ SC1-"	"τ SX"	"τ SB2-"	FP-)

5.5.4. One pointer per clause

Modelling this misconception involves replacing the rule applicable at disjunct nodes when the execution path is moving in a forward

direction. The student who has this faulty model of the Prolog interpreter imagines that if a clause has been previously used to satisfy a subgoal, it may not be used again, even on a fresh call. In the program:

p if a & b & c.

- a.
- a.
- b.

for instance, this would mean that on backtracking to the subgoal 'a' and resatisfying it, a fresh call to 'b' would fail since, according to this model of the interpreter, the fact 'b' has previously been used and is no longer available. To model this entails reflecting the behaviour in constructing the execution path from the CCS representation. When a disjunct is reached in a forward direction, the normal 'next-level' action is not always applicable. The determining factor is the previous traversal of a similar branch on an earlier occasion, even though this branch was approached from a different level. Seen in relation to the program above, the following diagram may help to illustrate this:

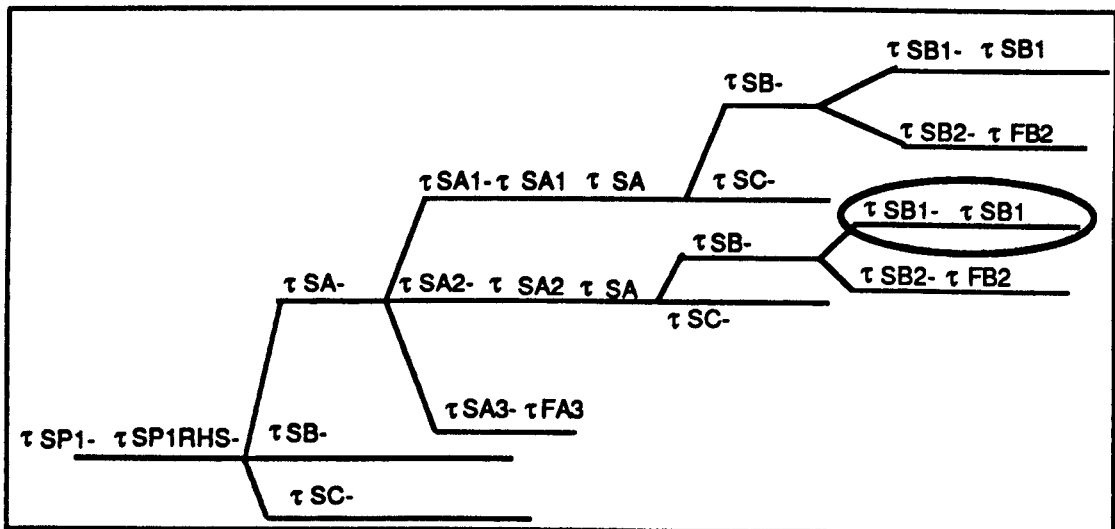


Fig.5.13 Once used, other instances of the branch are lopped off.

As can be seen from fig.5.14, initially the execution path of the program can successfully follow the branch beginning with 'τ SA1-' leading to disjunct 'τ SB-' and via 'τ SB1-' make a successful communication with

'τ SB'. After traversing this branch, which leads to an unsuccessful communication, denoted by 'τ FC1-' and having unsuccessfully tried the higher branch of 'τ SB2-', the execution path returns to disjunct 'τ SA-' and follows the higher level branch beginning with 'τ SA2-'. The execution path again successfully reaches disjunct 'τ SB-' and in normal search should again make a successful communication with 'τ SB' via 'τ SB1'. However, in terms of the tree, we can think of this second branch beginning with 'τ SB1-' as having been lopped off. Since the node 'τ SB1-' has been recorded previously, even though at a different level, the effect of the misconception being modelled is to assume that this branch cannot be successful. The execution path returns to the disjunct 'SA-' without successfully communicating with 'τ SB'. This behaviour is reflected in the rule:

```
(((position ?node)(direction forward)(type ?node disjunct)
  ((previously-used(next-level ?node))))
  ((position (up-levels ?node))(direction forward)))
```

The constraint of this rule, (previously-used(next-level ?node)) is a check on whether or not the next node in the execution path has been used before. If so, it cannot be used again, and communication with a higher level node is looked for, in this case, unsuccessful communication via τ SB2-. The function 'previously-used' checks to find if a similarly named node, even if at a different level in the tree, has been recorded in the list of positions in working memory. To do this it makes use of the list of positions recorded to date and the data from the CCS tree added at the initialisation to working memory. If it is found that the node has been previously recorded, then the action of the rule is to ignore that node and attempt a higher level node of that name. If the function fails, indicating that the next node has not been

recorded before, even at a different level, then this rule fails and the following one applies:

```
((position ?node)(direction forward)(type ?node disjunct)
  ((not(previously-used(next-level ?node))))
  ((position (next-level ?node))(direction forward)))
```

Revising the disjunct rules for when the execution path is moving forward completes the ruleset for modelling the 'onepointer per clause' misconception. Using this ruleset, for the program given above at the beginning of this subsection as an example, the system produces the following execution path:

(SP	"τ SB"	"τ SA-"	"τ SA3-"
"τ SP1-"	"τ SC-"	"τ SA2-"	"τ FA3"
"τ SP1RHS-"	"τ SC1-"	"τ SA2"	"τ FA-"
"τ SA-"	"τ FC1"	"τ SA"	"τ FP1RHS"
"τ SA1-"	"τ FC"	"τ SB-"	"τ FP1"
"τ SA1"	"τ SB-"	"τ SB2-"	FP-
"τ SA"	"τ SB2-"	"τ FB2"	"τ SP2-"
"τ SB-"	"τ FB2"	"τ FB-"	"τ FP2"
"τ SB1-"	"τ FB-"	"τ SA-"	FP-)
"τ SB1"			

thus mirroring the 'onepointer per clause' misconception being modelled.

5.6. Summary

In this chapter we have described the development of a production rule system used to produce correct and incorrect models of Prolog backtracking in a selection of variable free Prolog programs. The system does so by using information about possible program behaviours generated from a semantic description of those programs. The faulty models of the Prolog interpreter modelled here are in no way the only

ones formed by novices Prolog programmers. In addition to the examples given above, it is of course possible to reproduce the behaviour of models which reflect combinations of these misconceptions. Combining the ruleset for the 'try once and pass' with the relevant rules from the 'facts before rules' ruleset, or by combining the 'one pointer per clause' ruleset with the relevant rules from the 'redo from left' ruleset models instances where a student suffers from a combination of misconceptions. There are two principal criteria to be used in determining which misconceptions and combined misconceptions to model. The first, relating to the latter, is the question of being able to separate out the misconceptions involved. Where this is practical then it would be sensible to consider their inclusion. The second consideration, relating to both, is a matter of how likely they are to occur. Misconceptions or combinations of misconceptions which are met frequently enough to merit the expectation that some novices in a given group will have formed these particular models then merit their inclusion in the system both from a diagnostic point of view and in the long term view for reasons of efficiency. In the following chapter we discuss the use of these models in diagnosis and report on an empirical study which was undertaken as an evaluation of the work described in this and the preceding chapter.

Chapter Six

6. An evaluation.

As we have seen in chapter five, the properties of a production rule system have meant that with relatively small changes to the basic ruleset a range of faulty models of the Prolog interpreter can be generated from the semantic description of the program concerned. These properties stem from the requirements of a production rule interpreter that the behaviour being modelled must be described clearly in as general terms as possible and from the modularity inherent in viewing a process as a series of steps, each independent but affecting the final outcome by the order, type and frequency with which they are performed. The long term goal of developing such a system would be to use these properties to generate the individual execution models of Prolog backtracking formed by novices, even when these do not conform to the more common erroneous models. The construction of these latter models is a first step in exploiting the use of formal descriptions in diagnosing control flow errors and is one that demonstrates the potential of this approach.

The current goal has been to generate models of documented misconceptions from the given programs rather than pre-storing those misconceptions for each program involved. This has been achieved by using the semantic information generated from the given programs. As an evaluation of the system, it was used to diagnose the results of a second empirical study of novice Prolog programmers' models of backtracking. In this chapter we report on that empirical work and

discuss the results of computer analysis of the data, juxtaposing this with results obtained by a hand analysis of the same data.

6.1. A second empirical study

The immediate interest of the work discussed here centres on the strengths and weaknesses of the computational models used and any findings significant to the development of this research into the use of formal semantics in error diagnosis. The following sections give a brief outline of the experiment from which the data was obtained

6.1.1. Subjects taking part

The subjects participating in this experiment, which took place in summer 1988, were from a similar student background to those who took part in the previous empirical study reported in chapter two. They were distance learners taking an Open University degree course in psychology, attending a summer school week at Sussex University. They had completed a psychology project which involved programming in Prolog and were approximately at the same stage in their studies as third year undergraduate students. The project lasted two and a half days. It consisted of initially designing an algorithm which attempted to model a cognitive process, then implementing and debugging a Prolog program based on this algorithm. During the project they were free to ask for advice and help from the course tutors at any time.

Prior to the summer school week all these students had studied a short preparatory book-based introduction to Prolog [Eisenstadt 1987] as had

the students in the previous study, . This introductory booklet covers basic concepts of Prolog, i.e. facts and queries, the query interpreter, conjunctive queries, rules and database search. Students were expected to have completed the book-based course and accompanying exercises. Out of approximately seventy students who were available as subjects, thirty-four participated in the experiment, on a purely voluntary basis and analyses of the results were based upon the data obtained from these subjects.

6.1.2. Experiment design

This experiment was closely linked to the one undertaken twelve months previously, in that its immediate object was also to elicit from a group of novice Prolog programmers their predictions of the control flow in Prolog programs. Each subject was given six short programs and asked to describe what she or he believed would be the action of the Prolog interpreter in executing each program.

In contrast to the previous year, the experiment was designed as an on-line questionnaire and a complete set of the screen displays, which include the programs given, is included in Appendix C1. In order to restrict the area of errors which could be made, to those involving the search and backtracking behaviour of the Prolog interpreter and in some measure to constrain the range of interpretations of student errors, a variable-free subset of Prolog was used. The programs chosen were almost identical to those given to the subjects of the previous year in the paper and pencil experiment, the exception being problem six which in this experiment allowed for the 'rules before facts' misconception [Fung et al 1987]. As before, they were chosen to be as

simple as possible, whilst being sufficiently complex to allow the subject's answers to reflect her or his model of program execution.

The experiment was designed to run on a Macintosh, since these are the machines on which subjects had carried out their summer school work, so that they would feel reasonably at home using them. Preceded by an introduction and optional explanatory screen displays, the six programs were presented to the subjects in a series of windows, with a choice of buttons in each indicating the possible steps of the Prolog interpreter. Clicking on a particular button caused the words printed on it to appear in the rectangular area below the buttons, allowing the subject to keep track of steps they had already chosen. Subjects were able to edit their responses during or at the end of their answers to the problems. To make this clearer, we use the program:

```
a.  
b.  
b.  
p if a & b & c.
```

and some reproductions of the window which would be presented to the subject. On starting that particular problem, a subject would see the following screen with the buttons representing the options in predicting the steps of the Prolog interpreter.

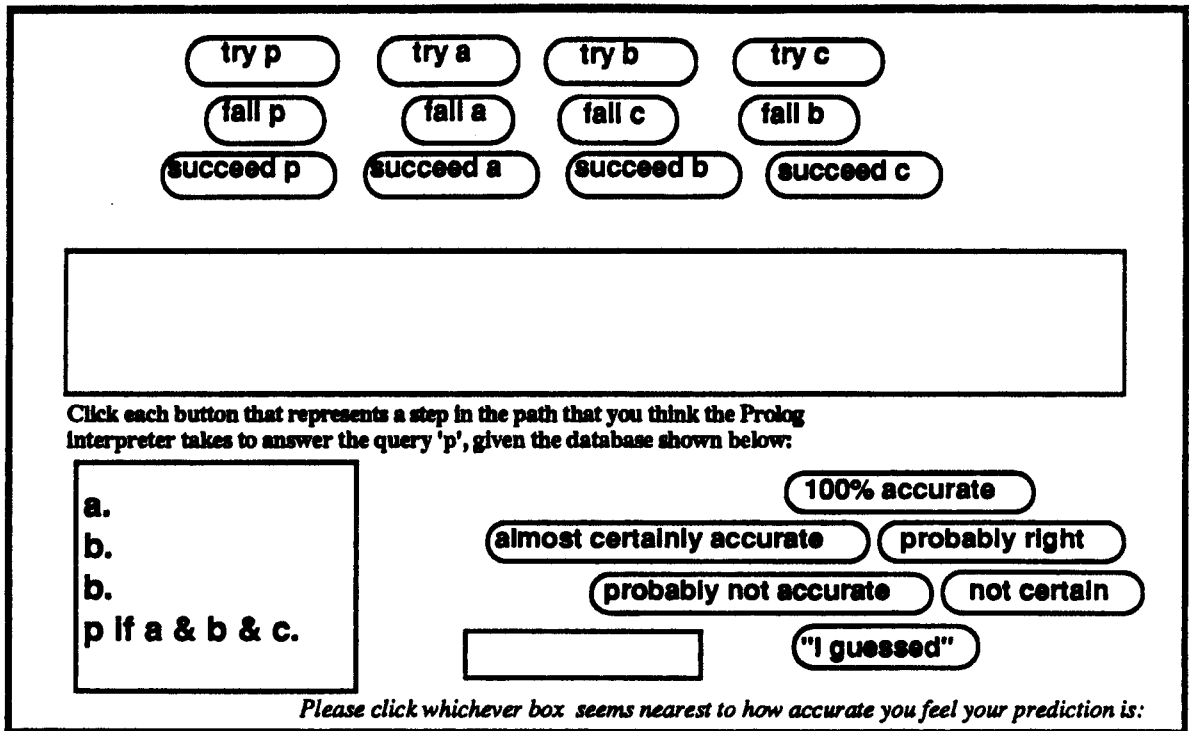


Fig.6.1 Layout of window shown to student

Below the buttons is the rectangle in which their answers are recorded and in the lower left hand corner is the given program.

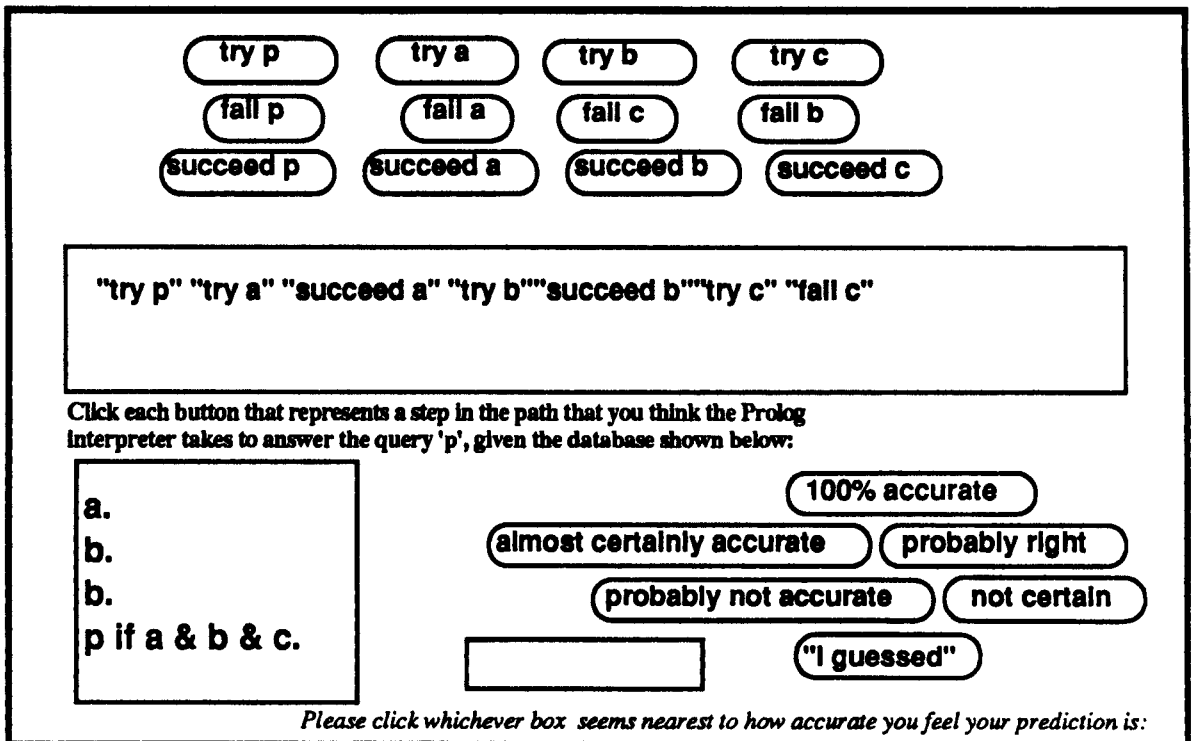


Fig.6.2 Window with student input

If the subject correctly predicts the execution of the program as far as the failure of subgoal 'c', by clicking the appropriate buttons, the screen would now look as above in fig.6.2. The lower right hand section of the window, with buttons labelled, for example, "100% accurate" or "almost certainly accurate", concerns 'confidence ratings' which will be explained and discussed in subsection [6.1.4].

Subjects were invited to take their time over completing the questionnaire, which was answered on an individual basis, without help from the experimenter. It was explained to each subject that the principal research interest served by the questionnaire was in obtaining the subject's predictions of the Prolog interpreter's execution of the various problems. The experimenter remained in the same room however, available to expand verbally on the explanatory screens at the beginning of the questionnaire and if the subject wished to do so, to discuss the problems after the questionnaire had been completed.

6.1.3. Problem design

The problems set in the experiment were designed to allow certain expected misconceptions to become apparent in the subjects' predictions. The interpretation of the faulty models of program execution which students' predictions provide, is an area which provides ample scope for further research. In some cases the errors which we have interpreted as symptomatic of certain underlying misconception may well support an alternative interpretation, the misconception 'try once and pass' being an example of this possible ambiguity. In practice we have stayed with the interpretation of the error types which we used in the previous empirical study and we

described in section [2.1.1]. These error types include the misconceptions which we have classed as 'redo from left', 'try once and pass', 'one pointer per clause', 'facts before rules' and 'rules before facts'. These last two misconceptions would only become apparent in either problem five or problem six of the problems set, since these were the only programs which included subgoals which had conditions, as can be seen from examining the programs in Appendix C1. The misconception 'one pointer per clause' would not be expected to appear in problem one since in the relevant program, on the failure subgoal 'b', it is not possible to resatisfy the first subgoal 'a', again, as can be seen by referring to Appendix C1. As mentioned above, the subjects were given the opportunity at the beginning of the experiment to work through an example problem to become familiar with the notation before proceeding with the six set problems.

6.1.4. Confidence ratings

The 'confidence ratings' mentioned earlier, akin to those used by Payne & Squibb [1986] were included to satisfy two interests. The first of these interests lay in whether or not the results of this would give a clear indication of a stable relationship between the level of confidence subjects felt in the accuracy of their answers and the actual accuracy of their answers. The second interest lay in whether or not in cases of faulty predictions it was possible to distinguish between those where subjects were confident in their 'faulty' model and those where there was an element of uncertainty which would indicate confusion rather than a relatively stable control flow model. Subjects were asked to click the confidence rating which they felt nearest to their own estimation of

their accuracy in each set of predictions. The ratings ranged from the highest level of confidence, "100% accurate" to the lowest level, "I guessed ". Their selected rating then appeared in the box to the left. As with the program predictions, if the subjects wished to do so, they were able to edit their selection at the time or before the end of the experiment, although in practice very few did so.

6.1.5. Data analysis

The results of the experiment were analysed by machine and then by hand. In both cases the basic classification of errors was made on the same principles. The subject's prediction of the interpreter's actions in each program was compared with the 'correct' prediction of the interpreter's behaviour in that problem. Where a difference was found, the subject's answer was then compared with the answer which would have been produced if the student had based her or his prediction upon one of the hypothesized faulty models of the interpreter described earlier. If the subject's answer fitted the pattern produced using one of these faulty models then it was noted as an error of that category, otherwise, in machine-analysis it was returned as a nil match and in hand analysis as an error of a complex sort,. This latter category will be discussed more fully in the section reporting on the results of the experiment. Since the main focus of this report is upon the use of automatic analysis of the data the following section is devoted to giving an outline of the method used and the structure of the processes involved.

6.1.6. Machine-analysis

The object of machine-analysing the data in this experiment was to evaluate the use of computational models constructed by the

production rule system described in the previous chapter. As input this system uses the semantic description generated from the programs in question. As we discussed in chapter four, producing this formal description of a Prolog program gives us a fine-grained picture of its execution, which can then be viewed as a search space of possible execution paths. Screen dumps of the semantic representation of each of the programs used in the experiment are included in Appendix D. As discussed in chapter five, applying the ruleset for normal Prolog search automatically traces the correct execution path through the search tree. If the student's predicted path does not correspond to this, then it is a question of determining which path has been taken. Rulesets for the misconceptions modelled are then applied to the CCS tree and the results used to determine if the student's prediction of the interpreter's path matches one of those.

The predictions from each subject taking part in the experiment were read into a datafile as she or he completed the problems. Each subject's datafile was numbered, but was otherwise anonymous. On-line analysis in real time was not possible due to hardware limitations and was postponed until data collection was completed. The data was analysed by a program based on a matching algorithm which compared the data obtained from each student for each of the six programs, with the output of the production rule system modelling the correct and incorrect execution paths for each of the same programs. As noted above this output was the result of applying the different rulesets to the search tree produced from the semantic representation of each program. The CCS representation of each particular execution path extracted in

this way from the search tree is automatically reduced to simplified subsets before being passed to the matching algorithm to be compared with the student data. i.e. the correct execution path derived from the search tree for the program:

p if a.
a.

```
(SP("τ SP1-"
("τ SP1RHS-"
("τ SA-"
("τ SA1-"
("τ SA1"
("τ SA"
("τ SP1RHS"
("τ SP1"
("SP-")))))))))))
```

would be reduced to ("try p" "try a" "succeed a" "succeed p").

As the output from the application of the ruleset for each model is supplied to the matching process, if a match with the student data is found it is recorded as an error related to that particular misconception. If no match is found, then a nil match is reported and the program continues the analysis with the next ruleset. Among the questions that were expected to be answered from using machine analysis were:

- (a) how well do the computational models of the particular 'faulty' execution paths correspond to those paths predicted by students who, on hand analysis of their data, would be diagnosed as exhibiting those particular control flow misconceptions?
- (b) has machine-analysis a useful role to perform in data-analysis of this sort, either replacing or augmenting hand-analysis?
- (c) does the computational modelling of known errors appear to be an efficient method of reducing the search space involved in a more sophisticated automatic diagnosis of students' errors?

In the following section the results of machine and hand-analysis are presented, prior to attempting to answer these questions.

6.2. Experiment results

The full results of machine-analysis of each problem for each student can be found in Appendix C2 and the full results of hand-analysis in Appendix C3. The number of answers analysed was 204. For machine-analysis, as explained in the previous section, if the student's prediction did not match the correct prediction for a problem, or any of the common errors expected, a nil match was reported, so the answer was counted as unanalysed, i.e it contained a complex error. In hand-analysing the data, if it was not possible to classify the student's prediction as correct or as one of the common errors, it was also classed as complex, not able to be classified with any certainty.

6.2.1. Answers analysed

Out of the total number of answers to be analysed, 164 were able to be analysed with reasonable certainty by hand, accounting for 80% of the total, while 112 were able to be analysed by machine, accounting for 55% of the total.

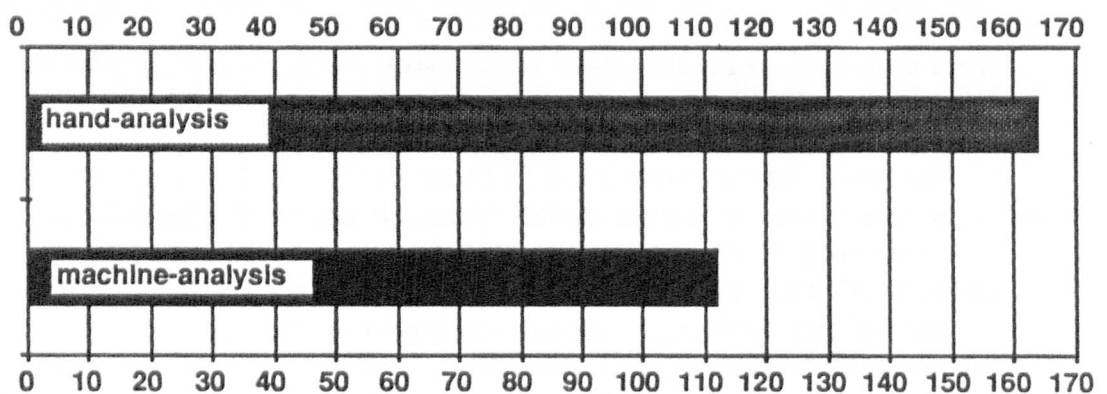


Fig.6.3 total number of answers analysed successfully

The number of answers successfully analysed by machine accounted for 68% of those successfully analysed by hand. The table below, fig.6.4, shows how this total was arrived at by comparing the number of answers successfully analysed for each of the thirty-four subjects.

The different success rate and the factors which contributed to it will be discussed fully in section [6.2.3]. Meanwhile it should be noted that the answers of five subjects account for a large part of the difference between the number of answers analysed successfully by machine and the number analysed successfully by hand.

subject no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
by hand	4	4	4	1	6	6	6	6	5	6	4	6	6	6	6	6	5
by machine	4	4	4	1	5	6	5	5	0	6	4	6	0	6	0	1	5

subject no.	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
by hand	6	1	6	4	6	6	5	6	4	6	5	5	1	6	4	2	4
by machine	6	0	6	1	6	6	0	6	0	6	0	1	1	6	2	0	3

Fig.6.4 Subject by subject breakdown of answers successfully analysed

6.2.2. Errors found

In the following figures, 'complex' errors indicate those cases mentioned above, in which the students' predictions could not be reasonably clearly accounted for and were not considered successfully analysed. In the machine-analysis results shown below, these 'complex' errors represent the 'nil' matches.

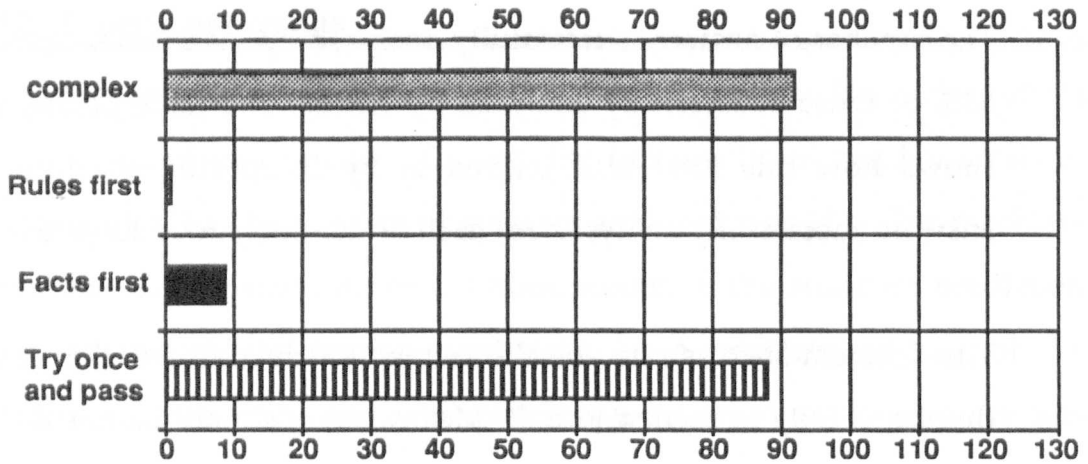


Fig.6.5 Breakdown of 190 errors found by machine-analysis

The errors found were not widely spread across the classes of errors expected. As is obvious from the above diagram (fig.6.5), the 'try once and pass' misconception, in which the student does not appreciate that any backtracking process takes place, was predominant.

It is also clear that in analysing the data by machine (see fig.6.5), the number of 'complex' errors, i.e. those reported as a nil match by the computer-analysis, was considerably higher than that resulting from hand-analysis, see fig.6.6 below. The factors which contribute to those different results are discussed below.

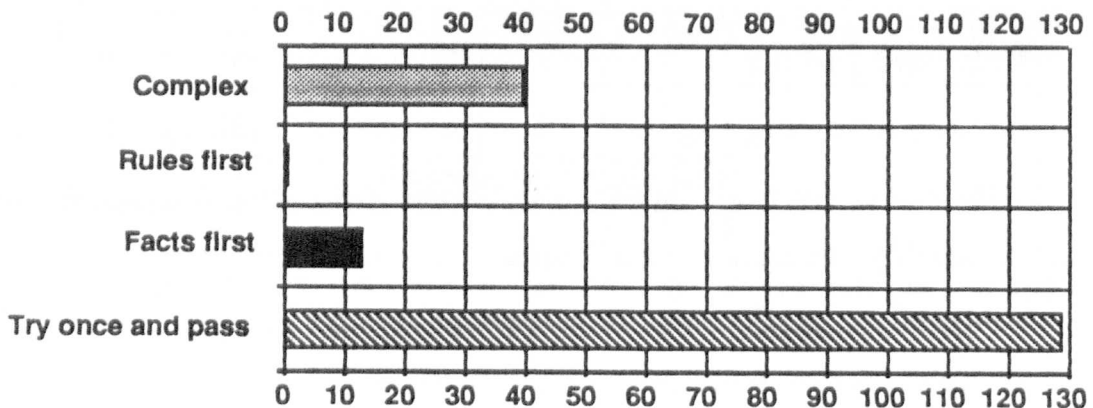


Fig.6.6 breakdown of 184 errors found by hand-analysis

6.2.3. Differences between analyses

The principal difference between the two sets of results lay in the total of errors classified as complex. In the case of machine analysis the only information available was that provided by the algorithm modelling the particular error, or combination of errors.

In cases where other factors distorted a student's prediction, even if there was a basic pattern of a common error in that prediction, the machine analysis could not successfully identify it.

The results of this can be seen in fig.6.7 overleaf, which compares the total number of errors made by each student, as analysed by hand and by machine. While machine-analysis reports a total of 92 complex errors (diagnosed as errors but not satisfactorily analysed), hand-analysis reports a total of only 40.

Factors accounting for the distortions which were unable to be detected by machine analysis seem related to experiment design and/or to information available for hand-analysis but not for machine-analysis. In the hand-analysed results in Appendix C3, the probable contribution of these factors to the higher number of successfully hand-analysed errors has been acknowledged in the following ways:

Hand analysis						Machine analysis					
error types						error types					
sub.	try once	facts first	rules first	complex	total	sub.	try once	facts first	rules first	complex	total
s1	3	0	0	2	5	s1	3	0	0	2	5
s2	3	0	0	2	5	s2	3	0	0	2	5
s3	3	0	0	2	5	s3	3	0	0	2	5
s4	0	0	0	5	5	s4	0	0	0	5	5
s5	5	1	0	0	6	s5	4	1	0	1	6
s6	5	1	0	0	6	s6	5	1	0	0	6
s7	5	0	0	0	5	s7	4	0	0	1	5
s8	5	1	0	0	6	s8	4	1	0	1	6
s9	4	0	0	1	5	s9	0	0	0	6	6
s10	5	1	0	0	6	s10	5	1	0	0	6
s11	3	0	0	2	5	s11	3	0	0	2	5
s12	5	0	0	0	5	s12	5	0	0	0	5
s13	4	1	0	0	5	s13	0	0	0	6	6
s14	5	0	0	0	5	s14	5	0	0	0	5
s15	5	0	0	0	5	s15	0	0	0	6	6
s16	5	1	0	0	6	s16	0	0	0	5	5
s17	4	0	0	1	5	s17	4	0	0	1	5
s18	5	0	0	0	5	s18	5	0	0	0	5
s19	0	0	0	5	5	s19	0	0	0	6	6
s20	5	1	0	0	6	s20	5	1	0	0	6
s21	3	0	0	2	5	s21	0	0	0	5	5
s22	5	1	0	0	6	s22	5	1	0	0	6
s23	5	0	0	0	5	s23	5	0	0	0	5
s24	4	1	0	1	6	s24	0	0	0	6	6
s25	5	1	0	0	6	s25	5	1	0	0	6
s26	3	0	0	2	5	s26	0	0	0	6	6
s27	5	0	1	0	6	s27	5	0	1	0	6
s28	4	0	0	1	5	s28	0	0	0	6	6
s29	4	1	0	1	6	s29	1	0	0	5	6
s30	0	0	0	5	5	s30	0	0	0	5	5
s31	5	1	0	0	6	s31	5	1	0	0	6
s32	3	1	0	2	6	s32	2	1	0	4	7
s33	1	0	0	4	5	s33	0	0	0	6	6
s34	3	0	0	2	5	s34	2	0	0	3	5
total	129	13	1	40	183	total	88	9	1	92	190

Fig.6.7 Comparison of error totals
by hand-analysis and by machine-analysis

(a) Layout

Where the layout of the test may have influenced the form of the student's answer in a way that was not anticipated in the experiment design, this is noted by the letter 'L' after the problem concerned. For

example, the hand analysis of answers given by subject fifteen looks as follows:

Subject 15	
Problem	result
1	T L
2	T L
3	T L
4	C L
5	T L
6	T L

This subject clearly followed the pattern of the 'try once and pass' misconception in each of the problems, so was classed as showing this error in the results. In addition she/he had in every problem inserted an extra step after the initial "try p", which in each case was "succeed p", as shown below in fig.6.9

Fig.6.8 hand-analysis results of subject 15.

In total this insertion of the step "succeed p" after the initial step to "try p" occurred with regularity in the answers of four subjects.

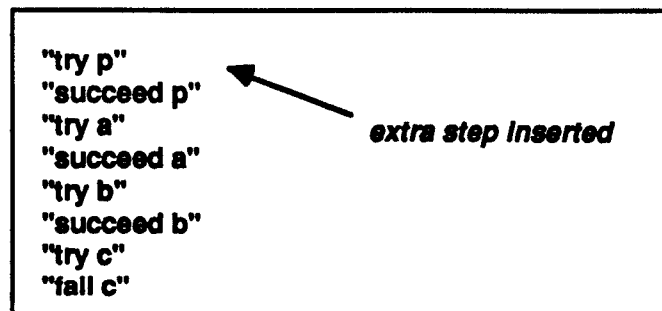


Fig.6.9 Prediction given by subject 15 for problem one

It seems plausible that this extra step may have been a result of constraints inferred by the subjects from the button layout. Having tried "p" the subject could well have felt that it was logical to record the finding of "p" in the data-base as successful, even though its conditions were not yet satisfied. This decision would seem reasonable since there was no button available which would equate to the state of 'holding' while the subgoal(s) were tried.

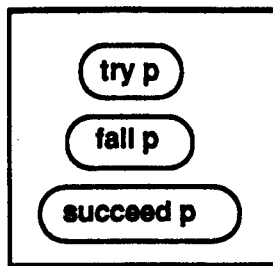


Fig.6.10 Layout of buttons for goal 'p'.

The probability of this reasoning process having taken place is supported at a very informal level by comments passed by a number of the subjects. While there had been no prior intention of collecting protocols of the subjects' comments as they answered the problems, it had, in the event been possible for the experimenter to observe and listen to most of the subjects as they worked. On working through an example problem, several mentioned the dilemma of deciding which button to click if no immediate "fail" or "succeed" seemed appropriate. Whatever its origin however, while the error 'try once and pass' was still able to be recorded by hand-analysis, due to this 'extra' step, the computer analysis was unable to record a satisfactory analysis.

(b) Meta-processing

The same difficulty for machine-analysis occurred where there appears to have been some element of either 'meta' processing by the student or belief that the Prolog interpreter possesses some 'meta-knowledge' [Fung et al 1987] about the final outcome of the program. In the hand analysis results, these cases have been noted by the letter 'M'. To illustrate this, if we look at subject 9, for example, she/he has in every case inserted the step "fail p" immediately after the initial step "try p", although the interpreter would not have taken this step until after all ways of trying to prove 'p' had failed.

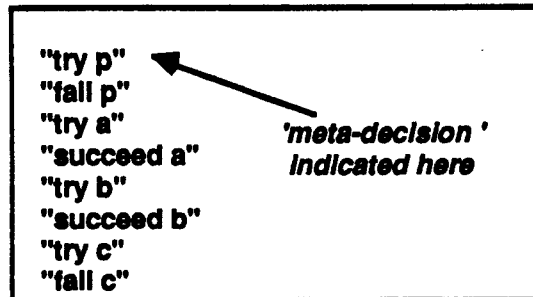


Fig.6.11 Prediction for problem one given by subject 9.

It cannot be ruled out that such answers may also have been influenced by the layout of the buttons as hypothesised above. However there seems a strong likelihood that also, or alternatively, the subject was attributing a 'meta-knowledge' to the Prolog interpreter reminiscent of the 'superbug' noted by Pea [1986]. It is also possible that she/he was identifying her/his mental execution of the program with the processes of the interpreter, the 'identity superbug' referred to by Taylor [1987]. Again, while accepting an open verdict on the origin of this false prediction, a 'try once and pass' misconception, for instance, was reasonably easy to see on hand-analysis of the answer. However, as the answer was complicated by this additional 'meta' factor, it could not be analysed successfully by machine.

(c) Abbreviations

There was yet another factor which accounted for some of the occasions where there existed a sharp difference between the results of hand and machine-analysis. In these cases it seemed the subject 'jumps' steps, i.e. abbreviates the prediction path. Examples of 'abbreviated' predictions given by subject twenty-eight are used to illustrate this, shown overleaf in fig.6.12.

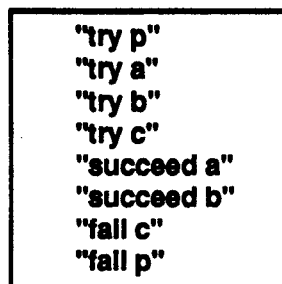
prob.1	prob.2	prob.3
"try p"	"try p"	"try p"
"try a"	"try a"	"try a"
"try b"	"try b"	"try b"
"fall c"	"fall c"	"fall c"
"fall p"		

Fig.6.12 'Abbreviated' answers of subject 28

Looked at during hand-analysis this can certainly be interpreted as answers from a subject who judges that the interpreter tries to prove each of the subgoals in order to prove the goal "p", and having tried once, reports failure of the goal without, apparently, any attempts to reprove the subgoals, i.e. a typical 'try once and pass' misconception. Yet there is no record of the outcome of the attempts to prove the subgoals, except the evidence implicit in the progression to the next subgoal and the ending of the stepping process when subgoal 'c' fails. This type of answer, when analysed by hand as showing evidence of one of the more common errors, is followed by the letter 'A'. This is intended to indicate that the analysis was made in the light of the fact that the student was very probably using an abbreviated annotation to describe the Prolog execution path. When analysed by machine, this type of answer was recorded as a nil match.

(d) Stacking

A final case which contributed to the higher number of mistakes successfully analysed by hand, was that of subject 16, whose predictions consistently followed the pattern shown overleaf, which is her answer to problem one.



```
"try p"  
"try a"  
"try b"  
"try c"  
"succeed a"  
"succeed b"  
"fall c"  
"fall p"
```

Fig.6.13 prediction given by subject 16

It is not clear exactly what is going on here. This particular subject did indicate from initial comments when working through an example problem that she thought the interpreter would call, i.e. 'try', all the subgoals first and then report on their success or failure. The self-assessment of accuracy for this subject (Appendix C4), confirms that this did seem to be her model of the interpreter, since she showed a high level of confidence in all her answers (level one for problems 1 to 4 and level two for problems 5 and 6). This does not however, on hand-analysis, hide the fact that she also showed that her model of the interpreter did not account for any backtracking process, i.e. that there was also a 'try once and pass' misconception. This subject's hand-analysed results (Appendix C3) have the letter 'S' added, indicating this model of 'stacking' calls by the interpreter. In machine-analysis this unusual pattern of prediction obscured the 'try once and pass' error, which consequently was not recorded in the five cases in which it happened.

In summary then, the differences between the results obtained by the two methods of analysis appear to have their roots in the differences between a method based on comparing expected models of control flow generated from the formal description of the programs used, i.e. the

current form of machine-analysis being evaluated, and a method, hand-analysis augmented by anecdotal evidence, where information about the students' actual performance was able to be reasoned about. This is significant in relation to the potential of the work reported here and will be discussed more fully in chapter seven. In the following subsection we consider the results of asking the participants in the experiment to assess their level of accuracy in predicting the execution path of the Prolog interpreter.

6.2.4. Confidence rating results

A complete record of each subject's self-assessment of her or his accuracy in each of the given problems is included in Appendix C4. As noted in section three, the interest in this data centred firstly on whether there was a reliable relationship between subjects' belief in their accuracy and their actual accuracy and secondly on whether this self-assessment would help to establish whether the models of the Prolog interpreter shown by the subjects were stable, i.e. reflected a subjects' confident belief in that model.

Work by Payne & Squibb [1986] in the domain of algebra, which influenced the inclusion of this self-assessment in the current experiment, did show a positive correlation between subjects' self-judgement of their performance and their actual performance. In contrast, the results in this experiment did not allow any clear-cut relation to be made. In general they did not indicate that self-

assessment of performance was a reliable indicator of actual performance.

In the following brief discussion of these results, we are looking at three basic levels of confidence indicated by subjects. Those most confident in their own accuracy, recording that they were 100% sure that their predictions were accurate are referred to as showing 'level one' confidence. Those recording that they were probably right or almost certainly right are referred to as showing 'level two' confidence. Subjects who recorded that they were uncertain or probably wrong are referred to as showing 'level three' confidence. This has been taken as the lowest level of confidence since none of the subjects used the option 'I guessed' to describe their predictions.

Corresponding to these 'confidence in their accuracy' levels, three levels were used to describe the actual accuracy of the subjects' answers. Level one accuracy is used to describe cases where the subject's prediction was correct. Level two is used to describe predictions which were not wholly accurate, but the pattern of which could be interpreted as belonging to one of the more common errors. Level three is used to describe predictions which were not accurate and not able to be interpreted. It was on this basis that the data was checked to see whether or not the levels of confidence expressed by the subjects tallied with the levels of actual accuracy registered for their predictions. The overall picture of the correlation between expected accuracy and actual accuracy was ambiguous. Out of a total of 204 predictions of the accuracy of the answer given (predictions for each of six problems from

34 subjects), 103 of those predictions coincided with the actual accuracy of their answer. Of the remaining 101 predictions, in 77 cases the actual accuracy was lower than predicted and in 24 cases higher than predicted. In the table below (fig.6.14), the abbreviation N/A is used where a particular outcome is not possible, e.g. a person predicting the highest level of confidence could not achieve any higher level of accuracy.

level of accuracy predicted	level of accuracy achieved					number of predictions
	same	one level lower	two levels lower	one level higher	two levels higher	
1	17	53	9	N/A	N/A	79
2	70	15	N/A	15	N/A	100
3	16	N/A	N/A	9	0	25
totals	103	68	9	24	0	204

Fig.6.14. Breakdown of total number of predictions

A scattergram based on the figures above shows these results more graphically.

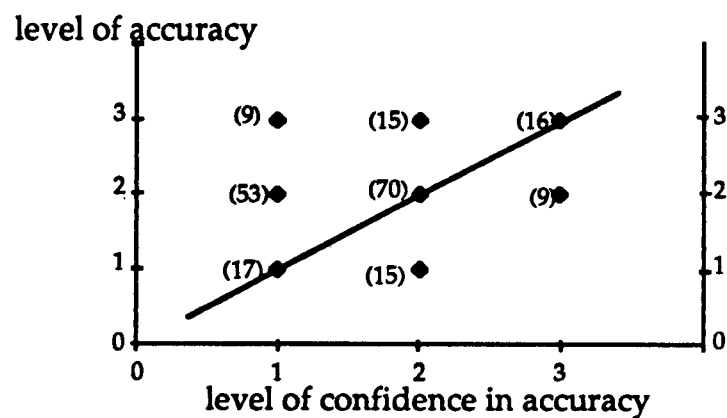


Fig.6.15 Relationship between confidence and accuracy

In the light of these findings one can justifiably conclude that it did not appear to be the case that subjects participating in this study had a reliable 'meta-view' of the accuracy of their own performance. The following diagram illustrates this with some individual cases. Out of the six students (shown below) who recorded level one confidence in their accuracy, only four actually achieved level one accuracy and in each case that was in only one out of the six problems set.

In each case the top row of numbers indicates the confidence level recorded by the subject for each of the six problems, the bottom row indicates the actual level of accuracy recorded.

s12						s22					
1	1	1	1	1	1	1	1	1	1	1	1
1	2	2	2	2	2	1	2	2	2	2	2
s20						s25					
1	1	1	1	1	1	1	1	1	1	1	1
1	2	2	2	2	2	1	2	2	2	2	2
s24						s33					
1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	2	3	2	3	3	3

Fig.6.16 Some individual cases showing differences between self-assessed confidence ratings of accuracy and actual accuracy

The second interest in the results of the 'confidence rating' was to consider whether it would give any insight into the consistency with which subjects held a particular model of the interpreter, i.e, if a student records "100%" confidence in the accuracy of each of her/his answers and in fact each of her/his answers reflects a 'try once and pass' misconception, then it would seem to confirm that the student is being consistent in her/his belief in that particular model of the interpreter.

However, this hypothesis could not be confirmed. In the answers of twenty-two out of the thirty-four subjects, each of those twenty-two subjects exhibited a consistent model of the interpreter in at least four out of the six given programs. As we have seen above though (fig.6.16), only six of those students expressed complete confidence in the accuracy of each of their answers. From these results it was not possible to use the self-assessed confidence rating as grounds for supporting the theory that the subject consistently believes in a particular model of the interpreter. Nevertheless, the results did reflect that there was a certain consistency in the appearance of a particular model of the interpreter, the 'try once and pass' model and this is considered in the following section.

6.2.5. Discussion of results

The main area of interest in the results of this experiment is that of evaluating the automatic analysis. This clearly and necessarily relates to the empirical data obtained in the course of carrying out the experiment. The range of more commonly met errors found in students' answers and successfully detected by machine-analysis in this experiment was determined by two factors. Firstly, the number of models that the system was programmed to generate for any given problem and secondly, the number of models represented by the students' predictions. As it happened, contrary to what one might expect, the range of models exhibited in the students' predictions in this study was narrower than that of the models able to be generated for the analysis.

This somewhat unexpected result can be regarded as the outcome of a combination of factors. One of the main objectives of the Open University summer school project which forms part of the psychology course, is to give students hands-on experience of using computers in cognitive modelling. The proportion of students who have had previous computing experience differs from year to year. In addition to the introductory reading and exercises in Prolog which they are expected to complete prior to summer school, the students have an overview lecture on Prolog from tutors at summer school. The algorithms and programs subsequently developed by students in the course of completing their projects vary enormously, both among the groups in a particular year and from year to year with different sets of students. To a large extent the models of Prolog formed by the subjects participating in the experiment reported here are influenced by this combination of variables, i.e. the extent to which students prepared themselves for the programming content of the project, the differing teaching styles of tutors and consequently differing emphases laid on certain aspects of Prolog, the type of programs the students chose to implement and the extent to which individuals in the groups involved themselves in the programming task.

It is difficult to say which of these was the determinant contributory factor, but the most striking feature of students' predictions in this study, was the consistency with which one particular model of the backtracking process was exhibited in students' answers. This consistency held true not only across the answers of each subject but also across the range of subjects. As noted in section [6.2.4] the model in question is one described earlier as typifying the 'try once and pass'

misconception. A student who has formed this model of Prolog reports failure of the program on the first failure of a subgoal of that program. She/he appears to be unaware of the exhaustive search that is conducted before the interpreter finally reports that a goal is unsuccessful.

The significance of the predominance of this 'try once and pass' error was that some computational models symptomatic of other backtracking misconceptions were effectively unevaluated. To compensate for this, the data from the similar experiment of the previous year (discussed earlier in chapter two) were analysed retrospectively by machine. The results of this analysis are discussed below in section[6.2.6].

6.2.6. Machine-analysis of 1987 summer school experiment

The hand-analysed results of the experiment conducted with the summer school subjects of 1987 which are essential to this discussion can be found in Appendix A. Figure 6.17 below shows the overall number of errors. The term 'unidentified errors' is used here to represent complex errors which were not able to be analysed with any degree of certainty.

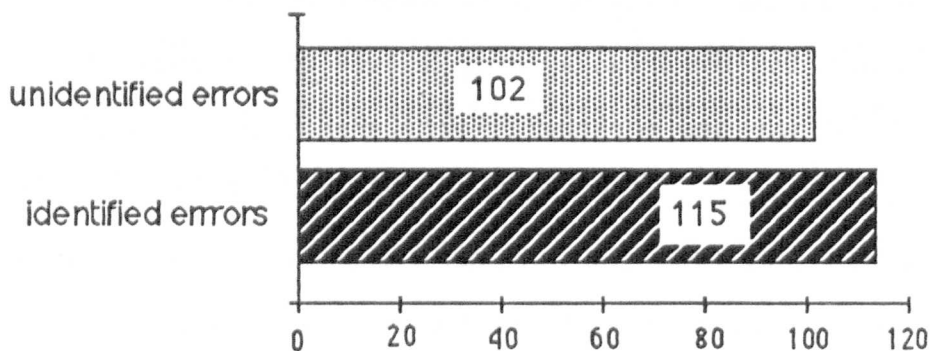


Fig.6.17 Total number of errors, hand-analysis,1987 data.

As explained in section[6.2.5] above, the objective in re-analysing this 1987 data by machine was to widen the evaluation of the machine-analysis program developed for use on summer school 1988 data.

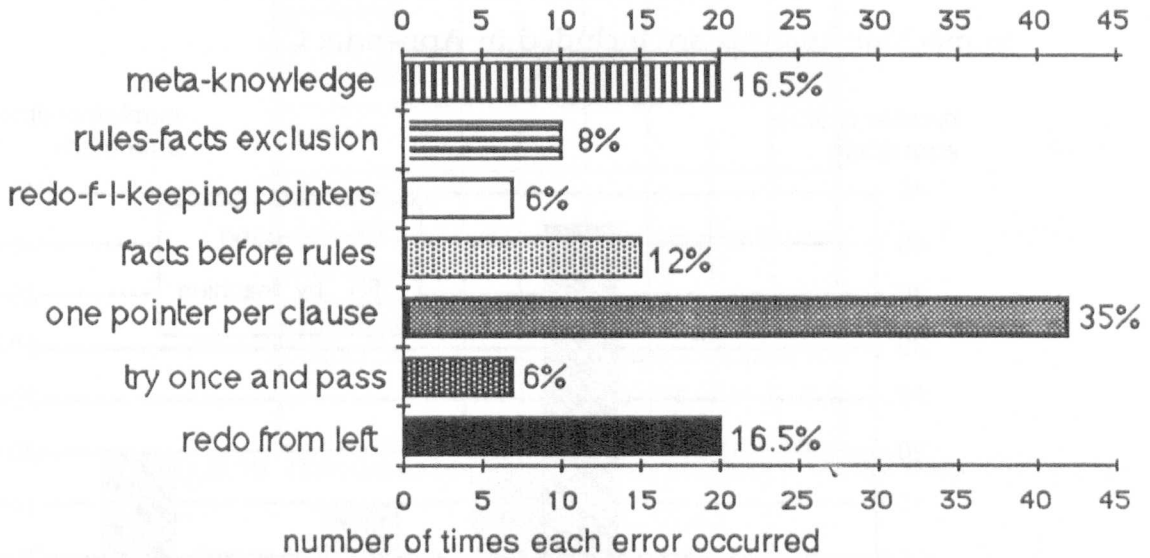


Fig.6.18 Breakdown of identified errors, hand-analysis, 1987.

In contrast to the narrow range of errors found in the 1988 study, data obtained by hand-analysis from the 1987 data showed that students participating in that experiment had made a large variety of errors.

The 'try once and pass' misconception appeared in the answers of only three subjects. The corollary to this was that the remainder of the thirty-two subjects showed an abundance of models of the backtracking process, the more easily identified of which are shown above in fig.6.18. The number of unidentified errors posed the same problems for machine-analysis as did their equivalent, the 'complex' errors of the 1988 study.

The computational models of errors were accurate in diagnosing particular errors where these occurred in isolation. In cases where the subject had compounded the error with other errors, this machine-analysis was unable to cope. The full results of subjecting the 1987 data to machine-analysis are included in Appendix C5.

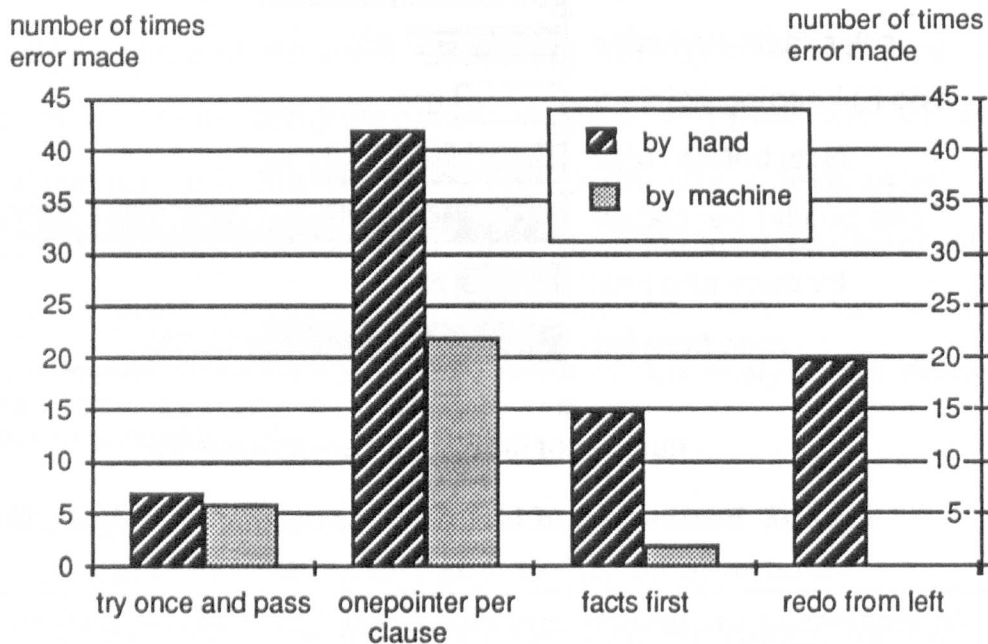


Fig.6.19 Differences, hand/machine-analysis, 1987 data

The diagram above illustrates the difference in error detection between hand and machine-analysis for the four misconceptions shown.

As would be expected in view of the restrictive effect of the 'try once and pass' error, machine-analysis detected six out of the seven occasions when this occurred in subjects' answers. In other cases the detection rate was very different from that achieved by hand-analysis.

subject	problem					
29	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledg						
	1	2	3	4	5	6

Fig.6.20 Hand-analysis results for subject 29, 1987 data

The 'redo from left' misconception was found in the work of six subjects, occurring in a total of twenty answers, yet it was not detected by machine-analysis on any occasion. In every case this particular error was combined with others, as in the example shown above (fig.6.20).

subject	problem						subject	problem					
6	1	2	3	4	5	6	8	1	2	3	4	5	6
redo-f-left							redo-f-left						
try-once							try-once						
one pointer							one pointer						
unidentifiable							unidentifiable						
facts first							facts first						
redo+pointers							redo+pointers						
rules-facts-excl							rules-facts-excl						
meta-knowledge							meta-knowledge						
	1	2	3	4	5	6		1	2	3	4	5	6

Fig.6.21 Results of hand-analysis of subjects 6 and 8, 1987 data

This was also the case to a lesser extent, for other errors such as the 'facts first' misconception, two examples of which are shown in fig.6.21 of the previous page.

The one pointer per clause misconception occurred in total in forty-two answers of the subjects who participated in the 1987 experiment. It was detected by machine analysis in only twenty-two of these cases, again because the error pattern was distorted by other errors.

subject	problem					
	1	2	3	4	5	6
5						
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

Fig.6.22 Results of subject 5, hand-analysis, 1987 data.

The exercise of analysing the 1987 data by machine served its purpose in that it tested the accuracy of the computational models of errors which had been developed using the CCS formalism. No incorrect positive diagnosis was made by the automated analyser. It also highlighted the point that awareness of likely mistakes and the ability to spot the more common errors quickly are important facets of diagnostic expertise, but as the only means of diagnosis, are insufficient. Both in the main

experiment described in this paper and in reviewing the data of the summer school 1987 experiment, machine-analysis using models of common errors exposed some of the limitations involved. These will be discussed below.

6.3. Machine-analysis comments

As explained earlier, the use of the models, generated and employed in the diagnosis of errors in a selection of Prolog programs was to serve two purposes. The immediate goal was to explore the use of a formal semantics in allowing the generation of these models from the Prolog programs in order to use them for diagnosing novices' control flow misconceptions, rather than pre-storing them for each particular program. The second goal was to evaluate the use of the models generated in this way as a means of reducing the search space in any further development of the production rule system. A future development of this system would attempt to generate models of the Prolog interpreter from the student input rather than matching student input with models generated by the system. Models such as those constructed in this work could then be usefully employed as heuristics in such future development. The questions asked of the models used in this evaluation are those we posed earlier. Do they produce accurate accounts of the misconceptions they are designed to model? Are they of practical use in augmenting or replacing hand-analysis in contexts similar to the one in which they were used? Are these models likely to be of use in reducing the search space in a diagnostic context?

The answers to these questions must be qualified ones. To the question of whether the use of formal semantics would allow the generation of accurate models of the misconceptions discussed, the answer seems clear. The results of the machine-analysis (Appendix C2), show that the models produced accurately reflected the predictions of those students whose answers, faulty or otherwise, clearly indicated that they were drawing on that particular model of the interpreter. The applicability of that accuracy is, however, restricted. As we have discussed earlier, there were cases in which the particular errors modelled were unable to be detected in the students' answers, although they were judged to be there in hand-analysis. This was due to additional 'distortions' of the expected model. These distortions arose from a variety of sources, as described in section [6.2.3], in some cases from factors perhaps introduced by the constraints of the experiment design, in others perhaps by the presence of additional misconceptions held by the subjects or even by the individual's approach to the task of recording her/his predictions. This restricted applicability becomes of more or less consequence according to the context of use.

It is this consideration of context which also gives rise to a qualification which must accompany the answer to the second question, 'Are they of practical use?' The answer 'yes' is justified because the use of the error models computationally generated for each of the programs represented a considerable reduction in the number of answers which needed to be hand-analysed. Sixty-eight percent of those answers successfully analysed by hand were also successfully analysed using the computational models of errors. It must, however, be noted that the

limits of the usefulness of such models can soon be reached. To add to the scope of these models by incorporating the ability to generate 'distortions' learned from the experience gained in one experiment would perhaps to some degree extend those limits, but essentially will do nothing to remove them. The experience of Stephen Payne [Payne & Squibb 1986] and others [Brown & Burton 1978] in the domains of algebra and arithmetic, respectively, has shown that clearly there are cases where students do apply 'buggy' models of procedures which account for a percentage of their errors. As their work has also shown, it is not so clear how far those 'buggy' models may be generalised across any given domain and how many 'buggy' models must be incorporated to account for all the student errors found. These limitations are not insignificant and will be returned to in a later discussion. In sum, the models such as those constructed and tested in this experiment are useful in a certain context. Used as the sole method of analysis, they lack analytical power unless a means of reasoning about the student's work is also incorporated in the system, but they have shown themselves to be useful in reducing the search task of analysing data in automatic analysis, i.e. the longer term context envisaged for their use.

6.4. Summary

In this chapter we have reported on a second empirical study. The data obtained from this study were used to evaluate the computational models of noted misconceptions of the Prolog interpreter. The hand-analysis of this data obtained from the 1988 summer school study, in combination with remarks informally noted while participants were working, produced several points of interest. It was apparent how

quickly subjects had formed specific models of the Prolog interpreter, which they often applied reasonably consistently to each of the programs given. For some subjects it was this 'consistent' or mechanistic aspect of the interpreter which had made most impression. Several participants passed remarks to the effect that whatever procedure was followed by the interpreter would necessarily be followed consistently. These subjects used this as a guideline in their predictions, even to the point of making such remarks as "well, in problem 'x' it would have taken those steps, so it would have to do the same here". It seems the experiment itself was playing a part in the learning process of these subjects, a phenomenon observed by [Hook, Taylor & du Boulay 1988].

The difference between the variety of models exhibited in the data obtained from the 1987 summer school study and the predominance of one particular model in the data obtained from this second empirical study in 1988 was also noteworthy. As discussed above, this must be seen as the result of several factors, rarely constant from year to year, which contribute to the formation of subjects' execution models of Prolog. This may be viewed as grounds for placing more emphasis on co-ordination of teaching material in an attempt to stimulate formation of the most suitable model as quickly as possible [Pain & Bundy 1987], [Brna, Bundy, Pain & Lynch 1987]. An interesting alternative has been put forward by [Hook, Taylor & du Boulay 1988] in the suggestion that for some students this initial forming of erroneous and sometimes inconsistent models of the interpreter can play an important part in the learning process.

The questions asked of the machine-analysis evaluation being undertaken were answered, albeit with qualifications. Given the extra data from the 1987 experiment, it was possible to use and evaluate the computational models constructed which were generated from the semantic descriptions of the programs in the experiment. Their use has shown that a formal semantics can successfully produce accurate computational models of Prolog backtracking errors.

Within the limits discussed in section [6.3], these models of common errors were able to fulfil a useful role in reducing the number of answers which needed more individual attention.

In the domain of algebra, Payne & Squibb [1986] point out that in Intelligent Tutoring, "using a pre-stored catalogue of mal-rules is unlikely to be a rewarding strategy".

This is no less true in programming. To achieve a consistently high rate of successful error diagnosis in this way entails an unacceptably high number of stored errors and fails to provide any flexibility of diagnosis in reaction to individual student input which does not accord with those errors.

The generation of models of more commonly found misconceptions of the Prolog interpreter and subsequent use of these in diagnosis is, however, a significant step in the path away from the prestored error

approach. It is an important step, demonstrating some of the potential of applying the central ideas of Milner's CCS to error diagnosis.

In the following chapter we take an overall look at what has been achieved in this current work. We will consider what we have learnt in the course of applying those ideas to Prolog and what contribution it has made to research in this area. We will outline how the work reported here can be situated within the larger framework of a tutoring system and how other aspects of the ideas put forward in Milner's CCS can be exploited in future research.

Chapter Seven

7. Conclusions

The goal of the work reported in this thesis was to explore the potential of a formal semantics in developing a diagnostic component of a tutoring module for novice programmers. Research undertaken in the course of that exploration focused upon a semantic-based diagnosis of novices' backtracking errors in Prolog. The formal semantics chosen was used to give a detailed description of a subset of Prolog As described in [section 4].this subset did not encompass the 'cut' or variables. The information generated from this description was used by a simple production rule system to diagnose a selection of control flow errors which we interpreted as being symptomatic of certain underlying misconceptions of the Prolog interpreter. The current research has focussed on investigating the viability of using a formal semantics as a means of resolving some of the problems faced in student modelling. In this chapter we briefly recapitulate what was achieved in the course of that research, discuss it within the wider context of intelligent tutoring systems and indicate some directions which future work in this area could usefully take.

7.1. Achievements

- * **An initial taxonomy of Prolog novices' control flow errors**
- * **A study of novices' models of the Prolog interpreter**
- * **Development of a machine-analysis tool**
- * **Development of a production rule description language**
- * **A computer-based empirical study**

The programming language Prolog was chosen as the domain in which to explore the use of formal semantics in student modelling for intelligent tutoring systems. An essential task was to define an area within the domain which was suitable for this exploration. As a first step, an initial taxonomy of Prolog novices' difficulties was undertaken. This taxonomy highlighted the difficulties which understanding control flow in Prolog presents for those first learning the language. An empirical experiment was then conducted which revealed in more detail the faulty models which novices form of the Prolog interpreter. The insight gained from this study indicated that this would be a fruitful aspect of the domain for study. Using a formalism based on Milner's CCS, a machine-analysis diagnostic tool was developed. This tool utilises information obtained from a detailed semantic representation generated from each Prolog program being used. A production rule system, for which a description language of the domain was developed, formed an important part of this machine-analysis system. The system was successfully used to identify and diagnose a number of the underlying misconceptions which novices have of the backtracking process in Prolog. In the following subsections we will look at each of these goals achieved and identify the key contributions each makes to student modelling in intelligent tutoring systems.

7.1.1. An initial taxonomy of Prolog novices' control flow errors

- * **Indicated the links between errors and misconceptions**
- * **Highlighted the need for more empirical research in this area**
- * **Provided a framework for investigating control flow errors in Prolog**

The compiling of an initial taxonomy of control flow errors in Prolog [Fung et al 1987] was one of the initial steps in the research reported in

this thesis. The taxonomy highlighted the need for empirical work to investigate the underlying causes of those errors more closely. It pointed out the possible links between the errors students make and the underlying misunderstandings from which these errors arise. By setting out, in an orderly way, the difficulties which the procedural aspect of Prolog presented for novices, the study provided a useful framework for discussion and investigation of Prolog control flow errors.

7.1.2. A study of novices' models of the Prolog interpreter

- * **Provided detailed information on students' models of Prolog backtracking**
- * **Provided insight into the links between errors and underlying misconceptions**
- * **Indicated the effects of vocabulary used in teaching programming**
- * **Highlighted the need to clarify the procedural nature of Prolog**

As a result of studying novices' control flow errors, it was decided to investigate in more detail the models which novices form of the Prolog interpreter. A study was made of students' expectations of Prolog program execution in cases where this would normally involve the backtracking process [Fung 1987].

The results corroborated existing research results in this area [Coombs & Stell 1975], [Taylor & du Boulay 1986]. They also produced examples of faulty models of the interpreter which were not anticipated but which appeared relatively consistently in the data obtained.

In a number of cases the erroneous predictions of program execution followed a pattern which indicated that they reflect an underlying

misunderstanding of Prolog control flow. The study strengthened the hypothesis that certain errors are linked to a faulty model of the interpreter.

The data obtained pointed to the likelihood that certain of the misconceptions noted in this study may well occur as a result of the vocabulary used in teaching Prolog. This was particularly noticeable in the case of references to 'rules' and 'facts'.

On many courses the procedural aspect of Prolog is not stressed in the first stages of learning. This first empirical study confirmed that students very quickly do begin to form some model of the procedural nature of Prolog, whether correct or incorrect. Early programs however, are often of the 'family relationship' kind, in which the solution is so related to everyday knowledge that an understanding of the underlying procedural behaviour of the interpreter is unnecessary.

The finding in this study that students begin at a very early stage to build models of program execution, strengthens support for the belief that in order to help student form correct models, this procedural aspect of Prolog should be clarified at the outset of learning the language [Bundy et al 1985], [Eisenstadt & Brayshaw 1987].

7.1.3 The development of a machine-analysis tool

- * **A way of formally describing procedural semantics of Prolog programs**
- * **Allows the generation of semantic descriptions of program behaviours**
- * **Offers a means of modelling novices' misconceptions**
- * **Facilitates the diagnosis of underlying misunderstandings**

A central part of the research is the development of a machine-analysis tool to be used for diagnostic purposes. To do this, the ideas of CCS are translated into the context of formally describing Prolog programs. Consequently these formal descriptions are used to generate the behaviours which those programs could produce. This development makes use of the information gleaned from the earlier empirical study and demonstrates the possibility of using a formal semantics to construct models of the Prolog interpreter.

Formal descriptions of the programs which had been selected for the empirical study were produced and from these, models were generated of the more common misconceptions which were encountered in that study. These models were then incorporated into the machine-analysis tool, which was subsequently successfully used to diagnose students' misunderstandings of the Prolog interpreter, using the data obtained from a second empirical study.

7.1.4. The development of production rule description language

- * **Provides a clear and precise description of the domain language**
- * **Allows production-rule modelling of program behaviours**
- * **Can be used to facilitate the automatic construction of diagnostic models**

An important component of the machine-analysis tool is a production-rule system. The system was developed as a means of constructing

correct and incorrect models of Prolog execution from the formal semantic descriptions generated from the given programs.

For this production-rule system it was necessary to develop a description language of the domain. The importance of the description language lies in the need to represent as cleanly and clearly as possible the actions taken in program execution.

The description language essentially forms the building blocks from which models of execution are constructed and as such is of significant importance in the present and future development of the diagnostic module. Its development is an important element in extending the system to incorporate machine-learning techniques.

7.1.5. Machine-analysis of a computer-based empirical study

- * **Showed the accuracy of models of misconceptions**
- * **Offered successful machine-analysis of students' predictions**
- * **Indicated factors affecting novices' formation of execution models**

A second empirical study of novices' models of the Prolog interpreter [Fung 1988] was undertaken to evaluate the machine-analysis tool which had been developed. The study was computer-based and the subjects' answers were captured for automatic analysis.

Analysis of the results by machine showed that some students had formed faulty models of the Prolog interpreter which reflected underlying misconceptions of program execution. These were able to be

detected and classified as such by the machine-analysis tool since they corresponded to models, generated from the formal descriptions of programs, which mirrored these misconceptions.

A comparison of the machine-analysis with a hand-analysis of the same results showed a smaller number of answers accurately diagnosed by machine. Inspection of the data reveals however that the answers of a relatively small number of subjects account for this difference. The principal factor which accounted for successful hand-analysis where in some cases machine-analysis was unsuccessful was the additional anecdotal evidence that was available to supplement hand-analysis.

This evidence showed that for a few subjects, the actual layout of the experiment design may have influenced their responses. For one or two others, their interpretation of explanations of Prolog control flow, given to them by their tutors in the course of learning the language, determined their expectations of program behaviour.

The data obtained from the study presented interesting indicators of the factors which contribute to the models of program execution which novices form. These results confirmed that the different manner in which Prolog is presented to the students and the structure of the programs which students encounter in their early learning of the language, play an important part in shaping their models of the interpreter [section 6.2.5].

7.2. The significance of these achievements for intelligent tutoring

As we discussed in section [1.4] a successful computer tutoring system will consist of a combination of components, which, used in conjunction with each other, assist the task of learning in the given domain. The research reported here must be considered as the development of one component within this larger framework. The goals achieved in developing the diagnostic component described here can be seen as contributing in three important ways to the ultimate goal of building intelligent systems for tutoring programming languages:

- in the contribution they make to the task of developing diagnosis in tutoring systems which will operate at a level that goes beyond the more superficial detection of errors.
- in the contribution which they make to the search for a means of student modelling that can be generalised across a range of programming languages.
- in the contribution which they make to existing empirical research into novices' difficulties, the understanding of which is a prerequisite to intelligent tutoring.

In the subsections below we look in turn at each of these areas of contribution.

7.2.1. A contribution to diagnosis in tutoring systems

- * **A move away from reliance on pre-stored information**
- * **A more flexible solution than existing meta-interpreters**
- * **A step towards analysis of underlying causes of errors**
- * **The potential of more flexible diagnostic modelling**

A major problem to be tackled in the context of student modelling is the difficulty of relating errors to underlying misconceptions which give rise to those errors.

The ideas put forward and developed in the present research are a means of moving away from the concept of 'mal-rules' and 'bug' catalogues which need to be prestored for each program being used for tutoring purposes. The initial exploration of the use of a formal semantics has provided, in the area in which it has been employed, a starting point for a new approach to the problem.

The machine-analysis described here uses the knowledge of Prolog search and of more commonly found misconceptions to reproduce models of these and check them against given data. The generation of the space of possible behaviours for the individual programs without the need for pre-wired meta-interpreters provides the possibility of linking back errors to underlying behaviour.

The production rule description language developed is a means of extending this process to building a model which reflects the underlying behaviour of the individual student's model of Prolog execution. As we pointed out earlier [section 1.3], most approaches to the student modelling problem have been in terms of pre-stored 'mal-rules' or 'bug catalogues'. Employing the production rule description language as a means of constructing models gives a potential flexibility which is not feasible in current systems.

7.2.2 A contribution to student modelling

- * **Offers a general solution to modelling programming languages**
- * **Provides a richer representation of domain knowledge for use in student modelling**

The case, put forward in this thesis, for using a formal semantics has been explored in a particular context, that of Prolog control flow. The work in this research has shown that generating the semantic representation of the programs being tutored has potential that is relevant to student modelling in an intelligent tutoring system for Prolog.

The use of a formal semantics is not however limited to this particular application. The contribution of formal semantics to this work must be seen in the wider context of its possible contribution to any tutoring system for programming languages.

7.2.3. A contribution to empirical research

- * **It has provided additional insight into Prolog novices' difficulties**
- * **It has given indicators for teaching strategies**
- * **It holds implications for the teaching curriculum**

The empirical studies of novices' expectations of Prolog control flow looked in detail at the erroneous models students have of program execution. These models reflected the difficulties which students have in understanding the procedural nature of Prolog execution. The

studies investigated the likely misconceptions which led to those faulty models and the findings have implications for the building of tutoring systems for programming languages. The studies have suggested that the link between the way in which the language is taught and the models of that language which the students form is a close one.

Data obtained from the empirical studies indicate that the vocabulary used in teaching the programming language and the early programs which are used to introduce the language must be carefully chosen. In constructing a tutoring system these factors must be taken into account and reflected both in the teaching curriculum and teaching strategies of the computer tutor.

7.2.4. Summary

In this section we have outlined the importance of the work in this thesis in relation to student modelling and its contribution to research in the field of intelligent tutoring systems. In the following section we discuss the immediate next steps which should be taken to extend this work and in conclusion look at the longer term work which can be undertaken on the basis of the concepts explored in this thesis.

7.3. Future directions

The immediate next steps in relation to the present research should be to extend the existing prototype of a diagnostic module for Prolog. In a broader context future work would be to investigate the use of formal semantics in intelligent tutoring systems for programming languages other than Prolog. We will look first at extensions which should be

made to the existing system before going on to consider the direction this work could follow on a wider scale than explored in this thesis.

7.3.1. Extensions to existing work

- * **Expand the model-building potential**
- * **Adapt the system to include variables**
- * **Consider cases of unification errors**
- * **Develop the system to be a real-time diagnostic tool**

The existing system can expand its potential for constructing models from student input, using the production-rule description language to do so. This extension would incorporate machine learning techniques to infer decision rules at the relevant choice points in the semantic descriptions. Where a faulty model of program execution constructed in this way is encountered relatively consistently, then this knowledge could be used to provide appropriate tutorial help for the user. This would also begin to address the question of multiple errors, which the current system does not tackle in a disciplined way.

In addition to this, the system should be extended to include variables. For the present purposes a variable free subset of Prolog is used, so the current system does not need to exploit the power of the CCS system to incorporate variables and value passing. An extension of this work would do so.

At an intuitive level, our miniature retailing system described in chapter three, for instance, could be enlarged to include 'variable' components. If we incorporate a variable component "Article1", this would offer as its actions an input action which would take a value,

('hat' for instance), which could be retrieved from its output action. The following paragraphs sketch a brief outline of how in CCS terms, variables can be seen as one more kind of machine capable of communication.

Initially we can envisage a variable simply as a machine which accepts a value through an input channel and if contacted can return that value through an output channel.

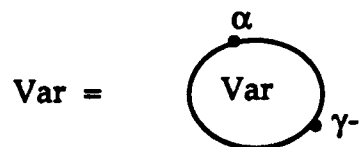


Fig.7.1. A first representation of a variable

i.e $\text{Var}(Z) \leq \alpha.\text{giveVal}(Z) + \gamma.\text{getVal}(Z)$.

where Z would be a variable, input to the machine Var via α , and retrieved via γ . The two associated actions consist of offering a value to Var at ' α ' or demanding a value from Var at ' γ '. However, in treating variables in Prolog we are modelling a more complex behaviour, since the instantiation of variables is an integral part of the unification process. To capture this, the Var machine will consist of three component machines. These relate to assigning a value for the variable (giveVal), holding the value in a register (holdVal) and outputting the value at the appropriate time (getVal). Each must have the possibility of the appropriate actions to communicate with each other.

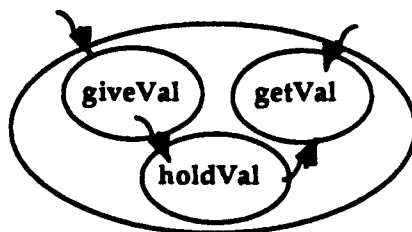


Fig.7.2. Components of a Var machine

In essence, the machine dealing with assignment, 'giveVal' would set the value of the variable:

$$[[X := E]] \leq \alpha_E \text{giveVal}_E(X)$$

Where X is a variable, and E the expression to which it evaluates, the action being to input the value of the variable to the machine 'giveVal' at ' α '. So if, for example, the variable 'H1' (perhaps short for HatShopClient no.1), is given the value of 'fred', and the variable 'H2' is given the value of 'gertrude', then we can describe this as follows:

$$[[H1 := fred]] \leq \alpha_{H1} \text{fred. giveVal}_{H1}(\text{fred})$$

and

$$[[H2 := gertrude]] \leq \alpha_{H2} \text{gertrude. giveVal}_{H2}(\text{gertrude})$$

The actions for holdVal, the component machine which holds the value of the variable, would be either to store a fresh value, or report the current value. The generic machine for this would be:

$$\text{holdVal}(Y) \leq \alpha X. \text{holdVal}(X) + \gamma. \text{holdVal}(Y).$$

i.e. $\text{holdVal}(Y)$, where Y is a value, offers a choice of actions, such that a new value can be input to it, ' $\alpha.X$ ' in which case it then becomes ' $\text{holdVal}(X)$ ', or the value it is holding can be returned at ' $\gamma.\text{holdVal}(Y)$ '. Note that this is a recursive machine.

Looking at one instance of this machine, we can take, for example, $\text{holdVal}_H(Y)$, where the value of the variable 'H' has been set at 'fred', then we would have the choice of two actions. A new value could be stored through the input action α , or the current value could be ascertained through the output action γ :

$$\begin{aligned} \text{holdVal}_H(\text{fred}) &\leq \alpha_H \text{ gertrude. holdVal}_H(\text{gertrude}) \\ &+ \gamma_H \text{ fred. holdVal}_H(\text{fred}) \end{aligned}$$

The getVal machine, when contacted, would output a value that it inputs from the holdVal component. Its action can be described as follows:

$$[[X]] \leq \alpha_X Z. \gamma Z$$

i.e. it can input (from holdVal) a value (Z) of a variable (X) at α , then return that value at γ , after which it has no more actions. An instance of this could be retrieving the value 'fred' stored in holdVal , of the variable 'H'.

$$[[H]] \leq \alpha_H \text{ fred. } \gamma \text{ fred}$$

Communication with the other components of a program would take place through the top-level 'Var' machine of which the three constituent machines, 'giveVal', 'holdVal' and 'getVal' are a part. Work on extension of the system to include variables is a direct step towards expanding the diagnosis of novices' errors to include faulty

models of unification. Since the unification process is another aspect of control flow which causes problems for novice Prolog programmers [Fung et al 1987] the system could then usefully be extended to cope with this.

A final immediate extension which must be mentioned concerns the implementational efficiency of the current development. At present the machine-analysis system operates on data after its collection. In implementing the current system, efficiency was not given priority, the paramount concern being to implement the concepts of the research. After incorporating the extensions discussed above, efficiency should merit a higher priority and the current system should be developed as a real-time diagnostic tool.

7.3.2 Longer term research - the role of formal semantics

The research reported in this thesis has demonstrated that formal semantics can be used to good effect in addressing the problems of student modelling. While the results have indicated that it is a promising approach to a difficult task, the work undertaken was confined to a subset of Prolog. This has been in itself a worthwhile enterprise.

The longer term goal must, however, be to demonstrate on a wider scale the contribution which formal semantics has to make to intelligent tutoring systems. Its use in a broader context, as a means of modelling programming languages, offers, as we discussed in chapter one, a fresh

approach to solving some of the difficulties which face present systems in terms of student modelling.

Much of the work which has been undertaken in the course of this research could, without undue difficulty, be translated into terms of another programming language. As Milner himself pointed out [Milner 1980], the ideas of CCS are intended by design to be adaptable for use in describing programs. The concepts involved are not limited in their applicability to any one particular language. A future extension of this research would be to apply the ideas of CCS to modelling a subset of a language other than Prolog. The work of Burstein [1985] gives a detailed account of the models which students form of variables when they begin to learn the programming language Basic. It would be interesting to investigate the use of CCS in constructing a formal representation of these models. The work necessary to extend the current system to include variables, for instance, would need little modification if applied to another language such as Basic. In that particular case, describing the semantics of a variable would in practice be a simplified version of the semantics of a variable in Prolog.

The essential point however is that the possibility of using formal semantics to model programming languages is not limited to the research area described in this thesis. The misconceptions of novice Prolog programmers were chosen as the area in which to undertake this initial exploration. Interesting and productive though this first investigation has proved to be, the ideas of CCS and the concept of using a formal semantics as a way of modelling programming languages has a

much wider application. The work in this thesis represents an initial exploration of the contribution of formal semantics to the task of student modelling in intelligent tutoring systems. The results of this exploration have indicated its potential contribution. Ultimately the goal of further research should be to exploit that potential in applying the ideas put forward in this thesis to other areas in the domain of tutoring programming languages.

References

- Adam, A. & Laurent, J. "A System to Debug Student Programs" in Artificial Intelligence 15, 75-122. 1980
- Adelson, B. "When Novices Surpass Experts: The Difficulty of a Task May Increase With Expertise" in Journal of Experimental Psychology: Learning, Memory and Cognition Vol.10. No. 3, 483 - 495. 1984
- Allison, L. "An Executable Prolog Semantics" in Algol Bulletin no. 50, 10-18. December 1983
- Allwood, C. "Novices on the Computer" in IJMM Studies 25, 633-658. 1986
- Anderson, J., Farrell, R. & Reiser, B. "Learning to Program in Lisp" in Cognitive Science 8, 87-129. 1984
- Anderson, J., & Reiser, B. "A Lisp Tutor, Greaterp (A goal restricted environment for tutoring and educational research on programming)" in Byte, 159-175. April 1985
- Anderson, J , & Jeffries, R. "Novice LISP Errors: Undetected losses of information from working memory" in Human-Computer Interaction vol 1, 107-131. 1985
- Apt, K., Van Emden, M. "Contributions to the Theory of Logic Programming" in Journal of the ACM vol.29, 3, 841-862. July 1982
- Beckman, L. "Towards a Formal Semantics for Concurrent Logic Programming Languages" in 225 Lecture notes in Computer Science Springer Verlag, 1986
- Beckman, L. "Towards an Operational Semantics for Concurrent Logic Programming Languages" Doctoral thesis, Uppsala University, 1987
- Beckman, L., Gustavsson, R. & Waern, A. "An Algebraic Model of Parallel Execution of Logic Programs" in Proceedings of the Symposium on Logics in Computer Science, Cambridge, 1986
- Bonar, J. & Cunningham, R. "Bridge: An intelligent tutor for thinking about programming" in Artificial Intelligence and Human Learning (Ed Self, J.) Chapman and Hall. 1988
- Bonar, J. & Soloway, E. "Pre-programming Knowledge: A major source of misconceptions in novice programmers" in Human-Computer Interaction vol 1, 2, 133-161. 1985
- Bratko, I. "Prolog Programming for Artificial Intelligence " Addison-Wesley, 1986.

- Brna, P., Bundy, A., Pain, H. & Lynch, L. "Programming Tools for Prolog Environments" in Advances in Artificial Intelligence (Ed. Hallam & Mellish) Wiley 1987
- Brna, P., Bundy, A., Dodd, T., Eisenstadt, M., Looi, C. K., Pain, H., Smith, B. & van Someren, M. "Prolog Programming Techniques" Research paper no. 403., Dept. of Artificial Intelligence, Edinburgh University 1988(submitted for publication in the special issue of Instructional Science on Learning Prolog: Tools and Related Issues).
- Brna, P., Brayshaw, M., Bundy, A., Elsom-Cook, M., Fung, P. & Dodd, A. "An Overview of Prolog Debugging Tools" Dept. Artificial Intelligence, Edinburgh University, Research paper no.398, 1988
- Brna, P. & Pain, H. "Observations of PROLOG novices" unpublished paper, Dept of Artificial Intelligence, Edinburgh University 1985
- Brown, J. & Burton, R. "Multiple Representations of Knowledge for Tutorial Reasoning" in Representation and Understanding (Ed Bobrow & Collins) Academic Press 1975
- Brown, J. & Burton, R. "Diagnostic Models for Procedural Bugs in Basic Mathematical Skills" in Cognitive Science 2, 155-192. 1978
- Bundy, A. "What Stories Should We Tell Prolog Students?" Working Paper 156, Dept. of Artificial Intelligence, Edinburgh, 1984
- Bundy, A., Pain, H., Brna, P. & Lynch, L. "A Proposed Prolog story" Dept of Artificial Intelligence Research Paper No. 283. Edinburgh University, 1985
- Bundy, A., Pain, H. & Someren van, M. "Prolog Programming Techniques" (forthcoming) Edinburgh, 1988
- Burton R. "Diagnosing Bugs in Simple Procedural Skills" in Intelligent Tutoring Systems D.Sleeman & J.S. Brown (Ed). Academic Press, 1982
- Burton, R. & Brown, J. "An Investigation of Computer Coaching for Informal Learning Activities " in Intelligent Tutoring Systems (Ed Sleeman & Brown) Academic Press, 1982
- Byrd, L. "Understanding the Control Flow of Prolog Programs" in Proceedings of the Logic Programming Workshop (Ed Tarnlund) Hungary, 1980
- Carbonell, J. "Mixed Initiative Man-computer Instructional Dialogue" Bolt Beranek & Newman Report, 1970

- Cerri, S., Fabbrizzi, M. & Marsili, G. "Trill - The Rather Intelligent Little Lisper" in Proc. of the AISB Conference on Artificial Intelligence and Education, Exeter, April 1983
- Cerri, S., Elsom-Cook, M. & Leoncini "TRILL: The Rather Intelligent Little Lisper" Centre for Information Technology in Education Report no. 48 Open University, 1988.
- Clancey, W. "The Role of Qualitative Models in Instruction". in Artificial Intelligence and Human Learning. (Ed Self) Chapman & Hall Computing, 1988
- Clocksins, W. & Mellish, C. "Programming in Prolog" Springer Verlag Berlin 1981
- Clocksins, W., & Mellish, C. "Programming in Prolog" Springer-Verlag Berlin (Second Edition) 1984.
- Coombs, M. and Alty, J. (Ed) "Computing Skills and the User Interface" Computers and People Series, Academic Press, London, 1981
- Coombs, M., Gibson, R. & Alty, J. "Learning a First Computer Language: strategies for making sense" in IJMM Studies 16, 449-486. 1982
- Coombs, M. & Stell, J. "A Model for Debugging Prolog by Symbolic Execution: the separation of specification and procedure" Dept. of Computer Science Strathclyde University, 1985
- Davis, R. "User Error or Computer Error?" in IJMM Studies 19, 359-376. 1983
- Dewar, A. & Cleary, J. "Graphical Display of Complex Information Within a Prolog Debugger in IJMM Studies 25, 503-521. 1986
- Dichev, C. & Du Boulay, B. "A Data Tracing System for Prolog Novices" Cognitive Science Research Report no.113. Sussex University, 1988.
- Du Boulay, B. & O'Shea, T. "How To Work The LOGO Machine" D.A.I Occasional Paper No. 4. Dept. of Artificial Intelligence, Edinburgh University, 1976
- Du Boulay, B. & O'Shea, T. "Teaching Novices Programming", in Computer Skills and the User Interface, (Ed Coombs & Alty) Academic Press, 1981
- Du Boulay, B. & Sothcott, C. "Computers Teaching Programming: an introductory survey of the field" in Artificial Intelligence and Education vol.1 (Ed Lawler & Yazdani) Ablex, 1987
- Du Boulay, B. O'Shea, T. & Monk, J. "The Black Box Inside The Glass Box: presenting computing concepts to novices" in IJMM Studies 14, 3, 237-249 1981
- Du Boulay, B. & Matthew, I. "Fatal Error in Pass Zero" in Behaviour and Information Technology 3, 109-118. 1984

- Eisenstadt, M. "A Powerful PROLOG trace package" in Proceedings of the Sixth European Conference on Artificial Intelligence ECAI-84, Pisa, Italy, 1984
- Eisenstadt, M. "Tracing and Debugging Prolog Programs by Retrospective Zooming" Human Cognition Research Laboratory Technical Report No. 17 Open University 1985
- Eisenstadt, M. "D309 Artificial Intelligence Project" Open University, 1987
- Eisenstadt, M. & Brayshaw, M. "The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming" Human Cognition Research Laboratory Technical Report No. 21 a, Open University, 1987
- Eisenstadt, M. & Brayshaw, M. "The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming" in Journal of Logic Programming 1988
- Eisenstadt, M. & Brayshaw, M. "An Integrated Textbook, Video and Software Environment for Novice and Expert Prolog Programmers" Human Cognition Research Laboratory Technical Report No.23 Open University, 1988
- Eisenstadt, M. & Lewis, M. "Errors in Interactive Programming Environment: causes and cures" Human Cognition Research Laboratory Technical Report No.4 Open University, 1983
- Elsom-Cook, M. "A User Interface for a Lisp Teaching System" in Proceedings of the Ergonomics Society Conference:the user interface, September 1983
- Elsom-Cook, M. "Design Considerations of an Intelligent Teaching System for Programming Languages" in Shackel (Ed) Human-computer interaction-INTERACT 84 Holland 1985
- Elsom-Cook, M. "Artificial Intelligence and Computer Assisted Instruction" Centre for Information Technology in Education Report No.4 Open University 1986
- Elsom-Cook, M. "Acquisition of computing skills " in Acquisition and Performance of Cognitive Skills (Ed Colley & Beech) 1988 forthcoming
- Fitting, M. "A Deterministic Prolog Fixpoint Semantics" in Journal of Logic Programming 2, 111-118. 1985

- Fung, P. "Novice Prolog Programmers: a consideration of their problems" Centre for Information Technology in Education Report, No.26 Open University, 1987(a)
- Fung, P. "Novices' Predictions of Prolog Control Flow" Centre for Information Technology in Education Report No.35 Open University, 1987 (b)
- Fung, P. "A Formalisation of Novices' Errors in Prolog Programs" Centre for Information Technology in Education Report no.50 Open University, 1988
- Fung, P. "Automated Diagnosis of Prolog Control Flow Errors: a first evaluation" Centre for Information Technology in Education Report no.81 Open University 1988.
- Fung, P., Du Boulay, B. & Elsom-Cook, M. "An Initial Taxonomy of Novices' Misconceptions of the Prolog Interpreter" Centre for Information Technology in Education Report no.27 Open University 1987
- Gilmore, D.J. "The Perceptual Cueing of the Structure of Computer Programs" Unpublished Doctoral Thesis, Psychology Department, University of Sheffield, 1986.
- Gilmore, D.J. "Programming Plans and Programming Expertise" in Quarterly Journal of Experimental Psychology (in press) 1988
- Gilmore, D. & Green, T. "Comprehension and Recall of Miniature Programs" in IJMM Studies 21, 31-48. 1984
- Goldstein, I. "The Genetic Graph" in Intelligent Tutoring Systems (Ed Sleeman & Brown) Academic Press 1982
- Gordon, M. "The Denotational Description of Programming Languages" Springer-Verlag 1979
- Gugerty, L. & Olson, G. "Debugging by Skilled and Novice Programmers" in Proceedings of the CHI '86 Human Factors in Computing Systems ACM 1986
- Hasemer, T. "An Empirically-based Debugging System for Novice Programmers" doctoral thesis, Open University, 1983
- Hartley, J.R. "The Design and Evaluation of an Adaptive Teaching System" in IJMM Studies 5, 421-436. (1973)
- Hennessey, S. "The role of Conceptual Knowledge in the Acquisition of Arithmetic Algorithms" Ph.D. Thesis University College London 1987

- Hennessy, M., & Milner, R. "Algebraic Laws for Nondeterminism and Concurrency" Internal Report CSR-133-83 Dept. Computer Science, Edinburgh University 1983
- Hoare, C. "Communicating Sequential Processes" Prentice-Hall International Series in Computer Science 1985
- Hook, K., Taylor, J. & du Boulay, B. "Redo 'try once and pass': the influence of complexity and graphical notation on novices' understanding of Prolog" Paper presented at PEG Third International Conference, Copenhagen 1988
- Jeffries, R. "A Comparison of the Debugging Behavior of Expert and Novice Programmers" presented at AERA Annual Meeting March 1982.
- Johnson, W. "Intention-based Diagnosis of Novice Programming Errors" Pitman 1986 London
- Johnson, W. & Soloway, E. "Proust: an automatic debugger for Pascal programs" in Byte 10, 4 179-190. 1985
- Johnson-Laird, P. & Wason, P. "A Theoretical Analysis of Insight into a Reasoning task" in Thinking: readings in cognitive science (Johnson-Laird & Wason Ed) Cambridge University Press 1977
- Jones, A. "How Do Novices Learn Programming?" Technical Report No.25 Computer Assisted Learning Research Group. Open University, 1981
- Jones, A. "How Novices Learn to Program: some protocol data" Technical Report No. 41 Computer Assisted Learning Research Group Open University 1984
- Jones, A. "Beginners mental models of a programming language" in The Computer Revolution in Education Harvester Press, Sussex, 1987
- Jones, A. "How Novices Learn to Program" in Proceedings of Interact'84 1984
- Jones, A., Scanlon, E. & O'Shea, T. (Ed) "The computer revolution in education" Harvester Press, Sussex, 1987
- Jones, D & Mycroft, A. "Stepwise Development of Operational and Denotational Semantics for Prolog " in Proceedings of the International Symposium on Logic Programming 281-288 1984
- Kahney, H. "An in-depth study of the cognitive behaviour of novice programmers" Human Cognition Research Laboratory Report No.5. Open University 1982
- Knuth, D. "The Remaining Trouble Spots in ALGOL 60" in Communications of the ACM vol 10, no. 10. 1967

- Lieberman, H. "An Example Based Environment for Beginning Programmers" Artificial Intelligence Laboratory, MIT, 1985
- Laurillard, D. "Evaluation of student learning in CAL" in Computers and Education 2 259-265. 1978
- Looi, C. "Automatic Program Debugging for a Prolog Intelligent Teaching System" Discussion paper 30 Dept. Artificial Intelligence, Edinburgh University, 1986
- Looi, C. & Ross, P. Automatic Program Analysis for a Prolog Intelligent Tutoring System" Dept.of Artificial Intelligence Research report No.307 Edinburgh University, 1986
- Looi, C. & Ross, P. "Debugging Prolog Programs in an Intelligent Tutoring System" Dept.of Artificial Intelligence Research paper No.308 Edinburgh University, 1987
- Looi, C. "Automatic Program Analysis in a Prolog Intelligent Tutoring System" PhD thesis, Edinburgh University 1987
- Lukey, F. "Comprehending and Debugging Computer Programs" in Computing skills and the User interface (Ed. Coombs & Alty) Academic Press 1981
- Mayer, R. "The Psychology of How Novices Learn Computing Programming" in Computing Surveys 13. 1. March, 1981
- Mellish, C. "Poplog Prolog" in A Portable Interactive Software Development Environment Cognitive Sciences Research Report No 100, Sussex University, 1988
- Milner, R. "A Calculus of Communicating Systems" 92 Lecture Notes in Computer Science Springer Verlag 1980
- Milner, R. "A Complete Inference System for a Class of Regular Behaviours " Internal Report CSR-111-82 Dept. Computer Science University of Edinburgh, 1982
- Milner, R. "Lectures on Calculus for Communicating Systems " NATO ASI series vol. F14 (Ed Broy) Control Flow and Data flow: Concepts of Distributed Programming, Springer Verlag 1985
- Murray, W. "Heuristic and Formal Methods in Automatic Program Debugging" in Proceedings of IJCAI 1985
- North, N. "A Formal Definition of Prolog" MSc.Dissertation, National Physics Laboratory Teddington 1986
- Norman, D. "Categorization of Action Slips" in Psychological Review, vol.88 no.1 Jan.1981

- Ormerod, T. "Content and representation effects with reasoning tasks in Prolog form" in Behaviour and Information Technology 1986
- O'Shea, T. "A Self-improving Quadratic Tutor" in Intelligent Tutoring Systems (Ed Sleeman & Brown) Academic Press, 1982
- O'Shea, T. & Self, J. "Learning and Teaching with Computers" Harvester Press 1983
- Pain, H. & Bundy, A. "What Stories Should We Tell Novice Prolog Programmers?" in Artificial Intelligence Programming Environments" (Ed. Hawley) Ellis Horwood, 1987
- Pask, G. "Styles and Strategies of Learning" in Br.J.Educational Psychology, 46 128-148. 1976
- Payne, S., Sime, M. & Green, T. "Perceptual Cueing in a Simple Command Language" in IJMM Studies 21, 19-29. 1984
- Payne, S., & Squibb, H. "Understanding Algebra Errors: the psychological status of mal-rules" Centre for Research on Computers and Learning Technical Report no.43, Lancaster University, 1986
- Pea, R. "Language Independent Conceptual "Bugs" in Novice Programming" in Journal of Educational Computing Research vol.2 (1) 1986
- Petre, M. & Winder, R. "Issues Governing the Suitability of Programming Languages for Programming Tasks" Research Note 88/7, Dept. of Computer Science, University College, London, 1988
- Plummer, D. "CODA: An extended debugger for Prolog" in Logic Programming vol.1. pp 496-511 Kowalski, R. & Bowen, P. (Ed) MIT Press 1988
- Rajan, T. "APT: A principled design for an animated view of program execution for novice programmers" Human Cognition Research Laboratory Report No. 19 Open University 1986
- Reiser, B., Friedmann, P., Kimberg, D. & Ranney, M. "Constructing Explanations from Problem Solving Rules to Guide the Planning of Programs" in Proceedings of ITS Montreal 1988
- Rich, C. Shrobe, H. & Waters, R. "Initial Report on Programmer's Apprentice for Lisp" MIT, 1976
- Rist, R. "Plans in Programming" in Empirical studies of programmers (Eds. Soloway & Iyengar) New York: Ablex 1986

- Robinson, J. "A Machine-Oriented Logic, Based on the Resolution Principle" in Journal of the ACM 12, 23-41 January 1965
- Ross, P. "Some Thoughts on the Design of an Intelligent Teaching System for Prolog" in AISB Quarterly No.62 Summer 1987
- Ross, P. "Teaching Prolog to Undergraduates" in AISB Quarterly 1982
- Sanderson, M. "Proof Techniques for CCS" Phd Thesis Edinburgh University, 1982
- Sanderson, M. "Bisimulation Techniques for CCS" Dept. Computer Science CSN-74 Essex University, 1985
- Scanlon, E. & Hawkrige, D. "Novice Physics Problem Solving Behaviour" in ECAI.84 Advances in Artificial Intelligence (O'Shea Ed.) Elsevier, N.Holland, 1984
- Scanlon, E. & O'Shea, T. "Educational Computing" Wiley Open University 1987
- Shapiro, E. "Algorithmic Automatic Debugging" Research Report 237 Yale University 1982
- Shneiderman, B. "Exploratory Experiments in Programmer Behaviour" in International Journal of Computer and Information Sciences 5 123-145 1976
- Shiel, B. "The Psychological Study of Programming" in Computing Surveys vol 13. 1. March, 1981
- Sleeman, D. & Brown, J. (Ed) "Intelligent Tutoring Systems" Academic Press, 1982
- Soloway, E., Bonar, J., & Ehrlich, K. "Cognitive Factors in Programming: An empirical study of looping constructs" in Communications of the ACM vol.26, 853-861. 1983
- Soloway, E., & Ehrlich, K. "Empirical Investigations of Programming Knowledge." in IEEE Transactions of Software Engineering SE-10, 5 1984
- Soloway, E & Iyengar, S (Eds) "Empirical Studies of Programmers", Ablex Publishing Corporation, Norwood 1986
- Someren van, M. "Misconceptions of Beginning Prolog Programmers" Memorandum No. 30 of the Research Project 'The Acquisition of Expertise' University of Amsterdam, 1984
- Someren van, M. "Beginners' Problems in Learning Prolog" Memorandum No. 54 Dept. Experimental Psychology University of Amsterdam 1985

- Someren van, M. "What's Wrong? Understanding Beginners' Problems with Prolog", in M.Eisenstadt (Ed) An Intelligent Computer Assisted Instructional System for Teaching Artificial Intelligence Programming, Human Cognition Research Laboratory Rep.no.26, Open University, 1988.
- Someren van, M. "Understanding students' errors with Prolog unification" Dept.Social Science Informatics VF Memo 102 Amsterdam University 1988
- Spohrer, J., Soloway, E. & Pope, E. "A Goal Plan Analysis of Buggy Pascal Programs" in Human-Computer Interaction vol 1, 2, 162-207. 1985
- Taylor, J. "Why Novices Will Find Learning Prolog Hard" (Ed.O'Shea) in Proceedings of ECAI Elsevier Science 1984
- Taylor,J. "Programming in Prolog: An in-depth study of problems for beginners learning to program in Prolog" D.Phil thesis, School of Cognitive Sciences,Sussex University 1987
- Taylor, J. & Du Boulay, B. "Studying Novice Programmers:why they may find Prolog hard" Serial No.CSRP.060 Sussex University, 1986 and in IJMM Studies 6 361-376. 1984
- VanLehn, R. "Felicity Conditions For Human Skill acquisition", Internal Technical Report, Xerox PARC, 1983.
- Weiser, M. "Program Slicing" in Proceedings of the Fifth International Conference on Software Engineering, IEEE , 439-449. 1981
- Wender,K., Weber, G. & Waloszek, G. "Psychological Considerations for the Design of Tutorial Systems" Psychologicsche Berichte Band 15 Heft 3 Unviersitaet Trier, West Germany, 1988
- Wenger,E. "Artificial Intelligence and Tutoring Systems" Morgan Kaufmann 1987
- White, R. "Effects of Pascal upon the Learning of Prolog - an initial study." Dept of Artificial Intelligence, University of Edinburgh, 1987.
- Wirth, N. "A Generalization of ALGOL" in Communications of the ACM vol.6 no.9 Sept., 1963
- Young, R. "The Machine Inside the Machine: users' models of pocket calculators" in IJMM Studies , 15, 51-85. 1981
- Young, R. & O'Shea, T. "Errors in Children's Subtraction" in Cognitive Science 5 1981
- Youngs, E. "Human Errors in Programming" in IJMM Studies vol 4, 361-376. 1974
- Zislis, P. "Semantic Decomposition of Computer Programs: an aid to program testing" in Acta Informatica 4, 245-269. Springer-Verlag, 1975

Appendix A1

**Booklet given to subjects taking part in
summer school experiment 1987**

Prolog experiment summer school

august 1987

Summer school August 1987

Thank you for agreeing to answer these questions. The following queries about Prolog and learning to program are planned to help us find out where people are likely to make mistakes and how to plan our teaching to make the job of learning Prolog as painless and fast as possible.

Before you start, make sure that you have read section 4.3 in the D309 Artificial Intelligence Project course book SUP 151774, and appendices A1, A2 and A3 .

Don't spend too long doing the answers, maybe half-an-hour at the most. Take your time over looking at how the example given below is set out, so that you understand the notation used for the answers, then go ahead and do your best with the questions. If you find any particularly difficult, don't hesitate to discuss them with one of us after you have attempted the answers.

Most of the questions are set out like the one on the following page. You are given a short program, and then asked to describe how the Prolog interpreter goes about answering the query put to it. Each little box in the strips represents one step in the Prolog search to answer the query. You can use as many of the boxes for each answer as you feel you need.

Example question and answer

PROGRAM

```

b.
a.
p if a & b & c.

```

QUERY

```

p.

```

(The first line of this program states that 'b' is true . The second line says that 'a' is true and the third line says that 'p' is true if it can be proved that 'a' and 'b' and 'c' are true. The query asks, " is 'p' true ?")*

Prolog goal search.

(the steps that Prolog takes to prove that 'p' istrue)

p try	a try	a succeed	b try	b succeed	c try	c fail
b try	b fail	a try	a fail	p fail	p try	p fail

This answer predicts that the Prolog interpreter will take these steps

```

try : p      - a rule
try : a      - this succeeds because a is a fact
try : b      - this succeeds because b is a fact
try : c      - this fails because c is not in the program
try : b      - this fails because there is no other b in the program
try : a      - this fails because there is no other a in the program
try : p      - this fails because there is no other p, so the query fails

```

If you find this notation difficult to follow, please come and ask one of us to go over it verbally with you. We will be happy to do so

Now try the following questions on the next pages. Read the program and the query, then think about how Prolog will go about answering the query (proving that 'p' is true). Write each step that you think Prolog will take to do this, in the boxes in the empty strips given below the program. You may not need to use all the boxes.

* see back page

Question 1.

PROGRAM

a.
b.
b.
p if a & b & c.

QUERY

p.

Predicted Prolog goal search

Question 2.

PROGRAM

a.
a.
b.
p if a & b & c.

QUERY

p.

Predicted Prolog search

Question 3.

PROGRAM

a.
b.
a.
p if a & b & c.

QUERY

p.

Predicted Prolog goal search

Question 4.

PROGRAM

a.
p if a .

QUERY

p.

Predicted Prolog goal search

Question 5.

PROGRAM

p if a & b & c.
 a if x.
 b.
 x.
 a.

QUERY

p.

Predicted Prolog goal search

Question 6.

PROGRAM

a.
 a.
 b.
 e.
 p if a & b & c.

QUERY

p.

Predicted Prolog goal search

Question 7.

Have you previously learnt any programming languages apart from Prolog ?

yes (If yes, which languages)

no

Question 8.

Have you had an opportunity to use Prolog on a machine prior to this course ?

yes no

Question 9.

Did you have time to work through

all most some

of the Prolog exercises in the D309 Artificial Intelligence Project course book,
SUP 15177 4, before you came to summer school ?

* re notation

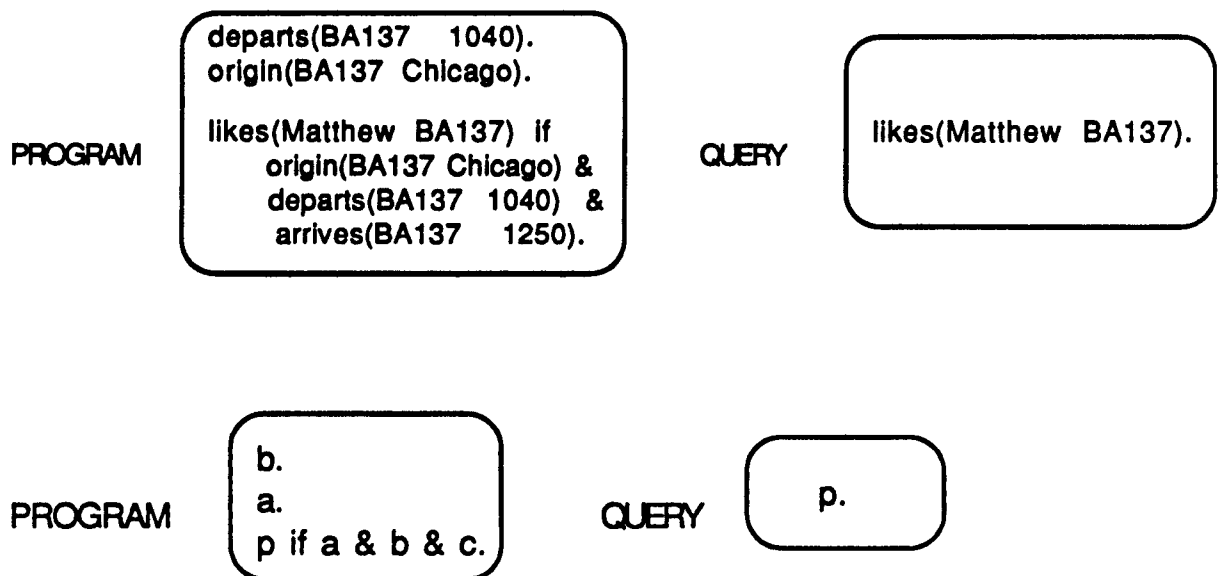
If, at first reading, the example program seems strange, it may help to compare it with the following program, which uses some of the database from the examples in the course book 'Artificial Intelligence Project' (section 2.2).

Both programs basically 'say' the same thing, i.e. that something is true (in one program the something is 'likes(Matthew BA137)', in the other it is 'p') if the three subgoals can be proved to be true (in one program the subgoals are 'origin(BA137 Chicago) & departs(BA137 1040) & arrives(BA137 1250)', in the other they are a & b & c).

In each of these two programs, the Prolog interpreter would take exactly the same steps to try and prove the query 'p' and the query 'likes(Matthew BA137)'.

It would try to match the query to a logical sentence in the database, in each case finding a match in a conditional logical sentence.

It would then query each conjunction of that matching sentence in turn, failing if it could not match all three conditions.



Appendix A2

The table overleaf shows the errors which each student showed evidence of in predictions of control flow in the six problems given. Also noted is each student's programming experience and whether or not each had completed the exercises set in the book-based introduction to Prolog course.

Table key

<u>Column heading</u>	<u>error</u>
A	<i>redo from left</i>
B	<i>try once and pass</i>
C	<i>one pointer per clause</i>
D	<i>unidentified</i>
E	<i>facts before rules</i>
F	<i>redo from left preserving markers</i>
G	<i>rules facts exclusion</i>
H	<i>meta-knowledge</i>
W	<i>indicates no programming experience prior to the course</i>
Y	<i>indicates no experience of Prolog prior to the course</i>
Z	<i>indicates that exercises set for the book-based introductory course in Prolog were not completed.</i>

SUBJECT	A	B	C	D	E	F	G	H						W	Y	Z	correct predictions
1	X		X			X	X							X	X	X	2
2				X	X		X	X							X	X	1
3							X								X		5
4			X											X	X		2
5			X	X			X							X			2
6				X	X											X	0
7	X		X	X	X	X		X						X			0
8				X	X			X						X	X	X	1
9																	6
10		X		X	X									X	X		0
11				X				X						X	X		1
12			X											X	X	X	3
13			X	X	X			X						X		X	1
14				X	X			X						X			0
15			X												X		3
16				X											X		1
17			X	X										X	X		4
18	X		X	X	X	X	X								X		1
19			X	X	X		X									X	1
20	X			X	X										X		0
21		X			X										X		1
22				X	X		X	X							X	X	1
23		X	X	X	X		X							X		X	1
24			X	X	X									X	X		1
25			X											X			2
26				X				X						X	X	X	1
27	X			X										X		X	1
28			X	X			X								X		1
29	X			X										X	X	X	1
30			X	X			X							X		X	1
31					X												5
32				X										X			0
	A	B	C	D	E	F	G	H						W	Y	Z	

Appendix A3

Individuals' results

subject	problem					
1	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
2	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
3	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
4	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6



error shown



multiple error (not inc.in total)

subject	problem					
5	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer		█	█			█
unidentifiable		█	█			█
facts first						
redo+pointers						
rules-facts-excl					█	
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
6	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable	█	█	█	█	█	█
facts first					█	
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
7	1	2	3	4	5	6
redo-f-left	█	█	█		█	
try-once						
one pointer			█		█	
unidentifiable	█	█		█	█	█
facts first					█	
redo+pointers			█		█	
rules-facts-excl						
meta-knowledge						█
	1	2	3	4	5	6

subject	problem					
8	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable	█	█	█		█	█
facts first					█	
redo+pointers						
rules-facts-excl						
meta-knowledge			█		█	█
	1	2	3	4	5	6

subject	problem					
9	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
10	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
11	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
12	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
13	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
14	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
15	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
16	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
17	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
18	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
19	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
20	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
21	1	2	3	4	5	6
redo-f-left						
try-once	█	█	█		█	█
one pointer						
unidentifiable						
facts first					█	
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
22	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable	█	█	█			█
facts first					█	
redo+pointers						
rules-facts-excl					█	
meta-knowledge		█	█			█
	1	2	3	4	5	6

subject	problem					
23	1	2	3	4	5	6
redo-f-left						
try-once		█				
one pointer			█			█
unidentifiable	█		█			█
facts first					█	
redo+pointers						
rules-facts-excl					█	
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
24	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer		█	█			█
unidentifiable	█	█	█		█	
facts first					█	
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
25	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer		█	█		█	█
unidentifiable						
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
26	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer						
unidentifiable	█	█	█		█	█
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge		█	█		█	█
	1	2	3	4	5	6

subject	problem					
27	1	2	3	4	5	6
redo-f-left		█				█
try-once						
one pointer						
unidentifiable	█	█	█		█	█
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
28	1	2	3	4	5	6
redo-f-left						
try-once						
one pointer			█			█
unidentifiable	█	█	█			
facts first						
redo+pointers						
rules-facts-excl					█	
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
	1	2	3	4	5	6
29						
redo-f-left	█	█	█			█
try-once						
one pointer						
unidentifiable	█	█	█		█	█
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
	1	2	3	4	5	6
30						
redo-f-left						
try-once						
one pointer			█			█
unidentifiable	█	█				
facts first						
redo+pointers						
rules-facts-excl					█	
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
	1	2	3	4	5	6
31						
redo-f-left						
try-once						
one pointer						
unidentifiable						
facts first					█	
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

subject	problem					
	1	2	3	4	5	6
32						
redo-f-left						
try-once						
one pointer						
unidentifiable	█	█	█	█	█	█
facts first						
redo+pointers						
rules-facts-excl						
meta-knowledge						
	1	2	3	4	5	6

Appendix A4

The following tables show for each error the percentage of problems in which a student made that particular error, i.e. in Table 1 below, subjects 8, 14 and 22 showed the 'meta-knowledge' error in sixty percent of the problems in which it was possible for this error to appear.

There is no table provided for the errors 'facts before rules' or for 'rules-facts exclusion', since these errors could only occur in one particular program, hence for each student such a table would simply show a hundred percent occurrence of the error.

Table 1. Percentage of problems in which the error 'meta-knowledge' occurred in students' predictions

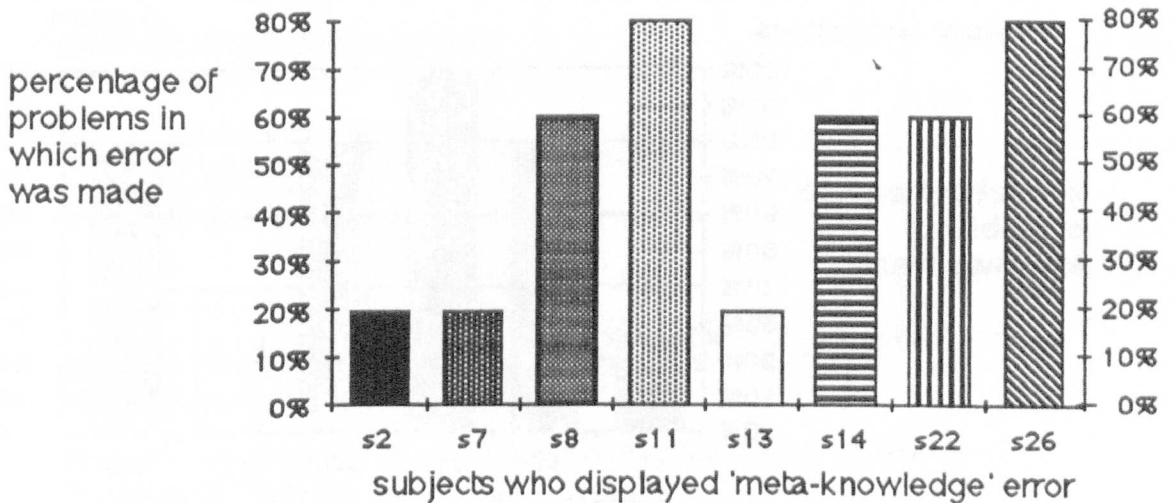


Table 2. Percentage of problems in which the error 'one pointer per clause' occurred in subjects' predictions.

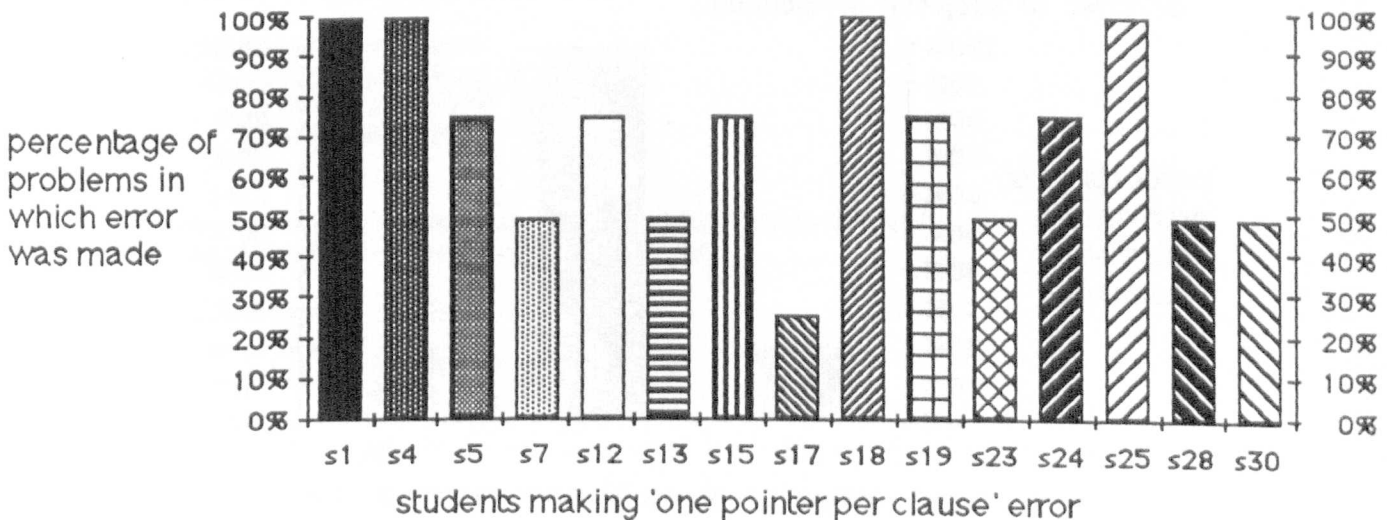


Table 3. Percentage of problems in which the error 'redo from left preserving pointers' occurred in each subjects' predictions.

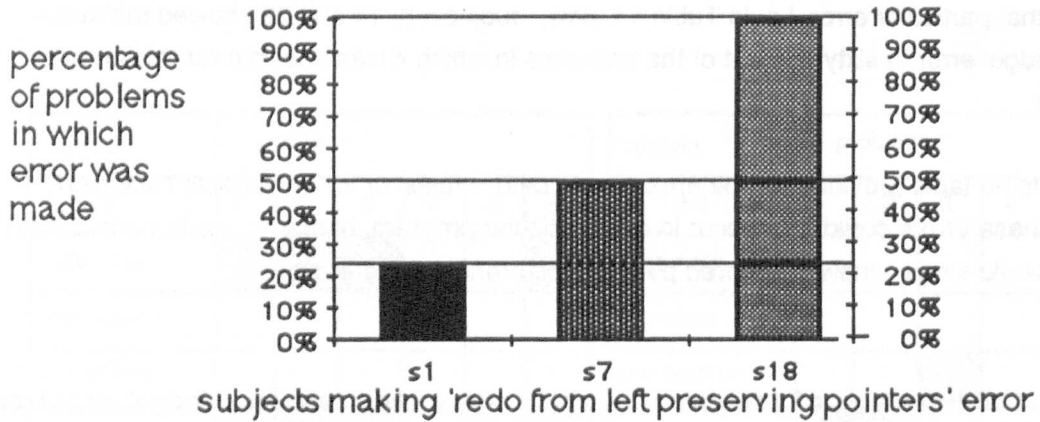


Table 4. Percentage of problems in which the error 'redo from left' occurred in subjects' predictions.

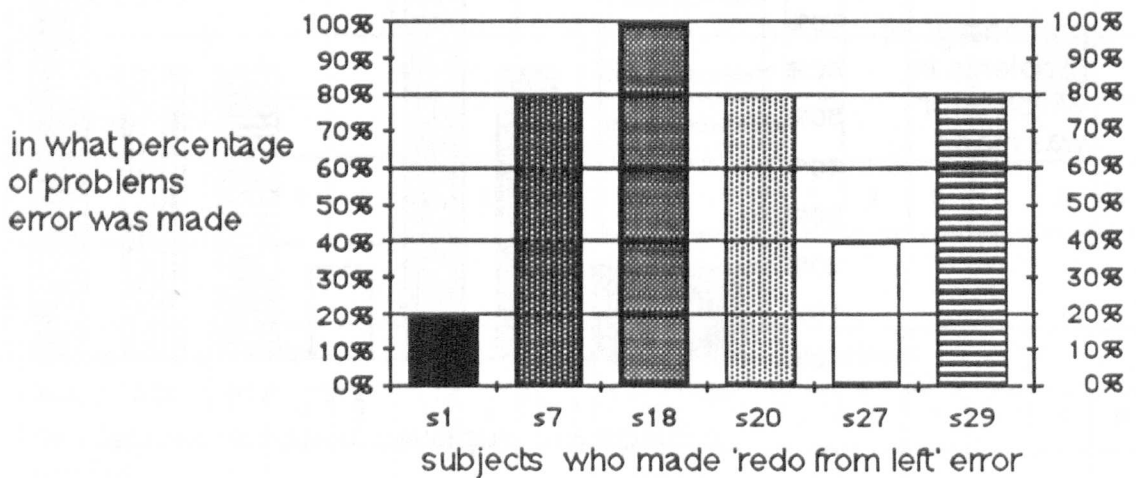
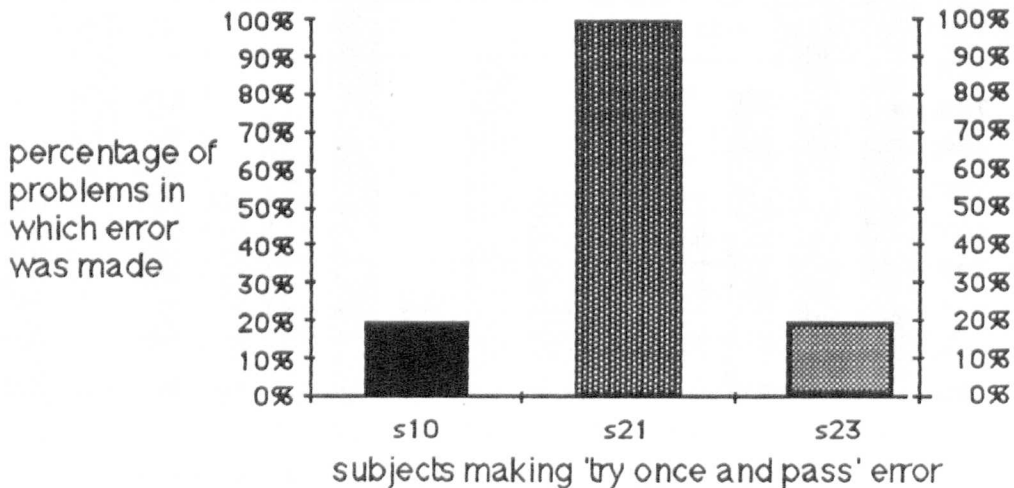


Table 5. Percentage of problems in which the error 'try once and pass' occurred in subjects' predictions.



;takes the head of the program component list on each pass and
 ;changes it into relevant ccs machine, depending on whether it is a query
 ;a fact etc.

```
(defun formalise (progbit)
  (cond((equal (car progbit) 'query) (query (name-of progbit)( clause-of progbit)))
        ((equal (car progbit) 'fact) (fact (name-of progbit)( clause-of progbit)))
        ((equal (car progbit) 'nomore) (nomore (name-of progbit)( clause-of progbit)))
        ((equal (car progbit) 'fail) (fail))
        ((equal progbit "CUT") (cut progbit))
        ((equal (car progbit) 'lhs-if-rhs) (lhs-if-rhs (name-of progbit)(clause-of progbit)))
        ((equal (car progbit) 'rhands1)(rhands1(name-of progbit)(clause-of progbit)
                                                (car(reverse progbit))))
        ((equal (car progbit) 'rhands2)(rhands2(name-of progbit)(clause-of progbit)
                                                (cadr(reverse progbit))(car(reverse progbit))))
        (t (rhands3(name-of progbit)(clause-of progbit)(caddr progbit)
                    (cadr(reverse progbit)) (car(reverse progbit))))))
```

```
(defun name-of(progbit)
  (cadr progbit))
```

```
(defun clause-of(progbit)
  (caddr progbit))
```

;this adds the list machines created by each program component, to the list of restrictions,
 ; unless one of them is a machine that we dont want restricted
 ; (initial machine and possible final states)

```
(defun update-restrictions(alist)
  (setf restrictions
    (remove-duplicates (append
                       (checkout alist) restrictions))))
```

;a member of that list of newly created machines might be a member of norestrictions
 ;so recurse down the list of newly created machines checking each new head

```
(defun checkout(alist)
  (cond((null alist) nil)
        (t (addn(checkit(car alist))
                 (checkout(cdr alist))))))
```

;if it is a member of the ones we dont want restricted
 ;leave it out, otherwise leave it in the list to be added to restrictions

```
(defun checkit(item)
  (cond((member (princ-to-string item) norestriction :test #'equalp) nil)
        (t item)))
```

;glue needed because of the various combinations
 ;which may be met in adding to restrictions list

```
(defun addn(a b)
  (cond((null a) b)
        ((and(atom a)(null b))(list a))
        ((null b) a)
        ((atom b)(list a b))
        (t(cons a b))))
```

;the various program components which may make up the program description
 ;the program is described in terms of queries, facts, nomore facts,
 ;fails, cuts, conjunctions.
 ;the next functions are used in changing these different components
 ;into the particular ccs machines

```
;a query
;e.g. (query "P" 2)
;(SP (+ (SP1- (+ (SP1 (SP-)) (FP1 (FP-))))
;      (SP2- (+ (SP2 (SP-)) (FP2 (FP-)))))
```

;if the program "P" is queried, the program machine 'SP' initially contacts
 ;the first P machine, SP1-
 ;if that contact is successful i.e. is recorded as SP1, then SP is successful,
 ;and its state is recorded as SP-
 ;if that contact is unsuccessful i.e. is recorded as FP1, then SP's state is
 ;failure and is recorded as FP-
 ;in the case of failure, the machine SP then contacts SP2-, which again, may be
 ;successful, or may not
 ;this continues until the machine SP does reach its successful state of SP- or
 ;until there are no more P machines to contact and thus SP's final state is failure i.e. FP-

```
(defun query(name clause)
  (update-restrictions (add2args "S" name))
  (append(add2args "S" name)
    (list
      (cons '+ (query2 name clause)))))
```

;each query machine will have two or more clauses (machines) to contact,
 ;depending on how many times it occurs in the program. A machine is created for

;each query clause, starting with the highest, until clause 1 is reached

```
(defun query2(name clause)
  (cond((equal clause 1)(update-restrictions (add4args "S" name clause "-")
    (list(append(add4args "S" name clause "-")
      (list(sporf name clause))))))
    (t (update-restrictions (add4args "S" name clause "-")
      (reverse(cons
        (append(add4args "S" name clause "-")
          (list(sporf name clause)))
        (reverse(query2 name (1- clause)))))))))
```

;a fact
 ;means that a successful communication can take place if it is queried
 ;e.g. if, say, there is a fact "A" clause one, i.e.(fact "A" 1),
 ;then it is represented as (SA1 (SA1-))

```
(defun fact (name clause)
  (update-restrictions
    (append (add3args "S" name clause)
      (add4args "S" name clause "-")))
  (append (add3args "S" name clause)
    (list(add4args "S" name clause "-"))))
```


;no fact
 ;on the other hand, if, say, there is no fact "A" clause two, i.e. (nomore "A" 2) then there
 ;is not going to be a successful communication with a query to "A"
 ;e.g. (SA2(FA2-))

```
(defun nomore(name clause)
(update-restrictions
  (append (add3args "S" name clause)
    (add4args "F" name clause "-")))
  (append (add3args "S" name clause)
    (list (add4args "F" name clause "-"))))
```

;lhs-if-rhs
 ;states that there is a rule
 ;if the query involved can only reach a successful state if it can
 ;succeed in its contact with all the items on the right hand side of the program
 ;e.g. if "P", in clause one, will only succeed if it can successfully contact "A" and "B"
 ;this is represented as
 ;(lhs-if-rhs "P" 1)
 ;and produces the machines
 ; (SP1 (SP1RHS- (+ (SP1RHS (SP1-))
 ; (FP1RHS (FP1-)))))
 ; i.e. the machine SP1 contacts the right hand side of the program (SP1RHS-)
 ; and this may result in success (SP1RHS) or failure (FP1RHS)
 ; which will determine the success (SP1-) or failure (FP1-) of SP1

```
(defun lhs-if-rhs (name clause)
(update-restrictions
  (append (add5args "S" name clause "RHS" "-")
    (add4args "S" name clause "RHS")
    (add4args "F" name clause "RHS")
    (add4args "F" name clause "-")))
  (append (add3args "S" name clause)
    (list (append (add5args "S" name clause "RHS" "-")
      (list
        (list '+
          (append (add4args "S" name clause "RHS")
            (list(add4args "S" name clause "-")))
          (append (add4args
            "F" name clause "RHS")
            (list(add4args "F" name clause "-"))))))))))))
```

; rhands describes the rule, (the right hand side of the program)
 ; at the moment it could be single, two-way or three-way
 ; e.g. it could consist of one, two or three conditions.

; a rule with one condition e.g. (rhands1 "P" 1 "A")
 ; the condition of the right hand side of P clause one is "A"
 ; this would produce the machines
 ; (SP1RHS (SA- (+ (SA (SP1RHS-))
 ; (FA (FP1RHS-)))))
 ; i.e. the right hand side of P clause one (SP1RHS) contacts machine SA-
 ; if this is successful, the right hand side of P clause one is successful
 ; (SP1RHS-) if not, then the right hand side of P clause one fails (FP1RHS-)

```
(defun rhands1(name clause cond)
  (cond((equalp cond "CUT")
        (list (cut cond)
              (append(add4args "S" name clause "RHS")
                    (list(endsorf name clause cond))))))
        (t (append(add4args "S" name clause "RHS")
                  (list(endsorf name clause cond))))))
```

;if the cut is one of the conditions on the right hand side
;then an extra machine is added to the list of machines
;denoting that it always succeeds e.g.(SCUT (SCUT-))

```
;(defun cut(progbit)
; (update-restrictions
; (append (add2args "S" progbit )
; (add3args- "S" progbit "-"))))
; (append (add2args "S" progbit )
; (list(add3args- "S" progbit "-"))))
```

;rhands2, a rule with two conditions e.g.(rhands2 "P" 1 "A" "B")
;produces the following ccs machines

```
;(SP1RHS
; (+ (SA- (+ (SA (SB- (+ (SB (SP1RHS-))
; (FB (FP1RHS-))))))
; (FA (FP1RHS-))))
; (SB-
; (+ (SB (SA- (+ (SA (SP1RHS-))
; (FA (FP1RHS-))))))
; (FB (FP1RHS-))))))
; i.e. the right hand side of P clause one (SP1RHS)
; succeeds if A and B can be contacted successfully(SP1RHS-)
; otherwise it fails (FP1RHS-)
```

```
(defun rhands2(name clause cond1 cond2)
  (let((rhandside (list name clause cond1 cond2)))
    (cond((member "CUT" rhandside :test #'equalp)
          (list (cut (car (member "CUT" rhandside :test #'equalp)))
                (append(add4args "S" name clause "RHS")
                    (list(each-and name clause cond1 cond2))))))
          (t(append (add4args "S" name clause "RHS")
                    (list(each-and name clause cond1 cond2))))))
```

;deals with combination of machines possible in rhands2,

;and success or failure

```
;(+ (SA- (+ (SA (SB- (+ (SB (SP1RHS-))
; (FB (FP1RHS-))))))
; (FA (FP1RHS-))))
; (SB- (+ (SB (SA- (+ (SA (SP1RHS-))
; (FA (FP1RHS-))))))
; (FB (FP1RHS-))))
```

```
(defun each-and(name clause cond1 cond2)
  (list '+
        (orsorf name clause cond1 cond2)
        (orsorf name clause cond2 cond1)))
```

;rhands3, a rule with three conditions e.g.(rhands "P" 1 "A" "B" "C")
 ;produces machines which allow "P" to succeed only if all three
 ; (SA-)(SB-)(SC-) machines on the right hand side are successfully contacted

```
(defun rhands3 (name clause cond1 cond2 cond3)
  (let((rhandside (list cond1 cond2 cond3)))
    (cond((member "CUT" rhandside :test #'equalp)
          (list(cut(car(member "CUT" rhandside :test #'equalp)))
                (append(add4args "S" name clause "RHS")
                        (list (inturn name clause cond1 cond2 cond3))))))
          (t (append(add4args "S" name clause "RHS")
                    (list (inturn name clause cond1 cond2 cond3)))))))
```

```
(defun inturn(name clause cond1 cond2 cond3 )
  (list '+
        (append(add3args- "S" cond1 "-")
                (list(sorf name clause cond1 cond2 cond3)))
        (append(add3args- "S" cond2 "-")
                (list(sorf name clause cond2 cond1 cond3)))
        (append(add3args- "S" cond3 "-")
                (list(sorf name clause cond3 cond1 cond2 )))))
```

;fail e.g.("FAIL") is represented as a fail machine which will
 ;only communicate unsuccessfully e.g. (SFAIL(FFAIL-))

```
(defun fail()
  (update-restrictions
   (append (list(intern(concatenate 'string "S" "FAIL" "-"))
            (list(intern(concatenate 'string "S" "FAIL"))
              (list(intern(concatenate 'string "F" "FAIL"))
                (list(intern(concatenate 'string "F" "FAIL" "-"))))))
           (append (list(intern(concatenate 'string "S" "FAIL"))
                   (list(list(intern(concatenate 'string "F" "FAIL" "-"))))))))
```

;the following functions do the various glueing together of machines

;sorf (success of program or failure) produces the disjunction
 ; machines representing the
 ;successful or unsuccessful outcomes of contact e.g. +(SA2(SA-)) (FA2(FA-))
 ;i.e. machine SA2 makes contact successfully (SA2), so SA reaches the success state SA-
 ;or machine SA2 fails to make successful contact (FA2), so SA fails (FA-)

```
(defun sorf (name clause)
  (update-restrictions
   (append (add3args "F" name clause)
           (add3args "S" name clause)
           (add3args "S" name "-")
           (add3args "F" name "-")))
  (list '+
        (append(add3args "S" name clause)
                (list(add3args "S" name "-")))
        (append(add3args "F" name clause)
                (list(add3args "F" name "-")))))
```

```
; endsorf (end success or failure) glues the bits to show success or fail of right hand side
; e.g. (SA- (+ (SA (SP1RHS-))
;      (FA (FP1RHS-))))
```

```
(defun endsorf(name clause cond)
  (update-restrictions(add3args- "S" cond "-"))
  (append (add3args- "S" cond "-")
    (list
      (list '+
        (success name clause cond)
        (failure name clause cond) )))))
```

```
;glue (SA (SP1RHS-))
```

```
(defun success( name clause cond)
  (update-restrictions (add2args "S" cond))
  (append (add2args "S" cond)
    (list
      (add5args "S" name clause "RHS" "-"))))
```

```
;glue (FA (FP1RHS-))
```

```
(defun failure(name clause cond)
  (update-restrictions
    (append (add2args "F" cond)
      (add5args "F" name clause "RHS" "-")))
  (append (add2args "F" cond)
    (list
      (add5args "F" name clause "RHS" "-"))))
```

```
;sorf (succeed or fail) glues all the bits together for each section of rhands3
```

```
(defun sorf(name clause cond1 cond2 cond3 )
  (update-restrictions (add2args "S" cond1))
  (list '+
    (append
      (append (add2args "S" cond1)
        (list (either name clause cond2 cond3))))
      (failure name clause cond1)))
```

```
;either (either first condition or second condition)
;glues the bits together for each section of rhands2
;hands one section at a time to orsorf(succeed or fail)
```

```
(defun either(name clause cond2 cond3 )
  (list '+
    (orsorf name clause cond2 cond3 )
    (orsorf name clause cond3 cond2)))
```

;orsorf (succeed or fail) takes each condition of either
;and glues the possible contacts

```
(defun orsorf(name clause cond2 cond3)
  (update-restrictions (add3args- "S" cond2 "-")
    (append(add3args- "S" cond2 "-")
      (list
        (list '+
          (append(add2args "S" cond2)
            (list (endsorf name clause cond3)))
            (failure name clause cond2 ))))))))
```

;the following functions do all the glueing at the string level,
;of turning the program names and clause numbers into machines
;e.g. "P" clause one, into SP1

```
;e.g. (add2args "S" "P") produces machine SP
(defun add2args (a b)
  (list(intern(concatenate 'string a b))))
```

```
;(add3args "S" "P" 1) produces (SP1)
(defun add3args (a b c)
  (list(intern(concatenate 'string a b (princ-to-string c))))))
```

```
;(add3args- "F" "P" "-") produces(FP-)
(defun add3args- (a b c)
  (list(intern(concatenate 'string a b c))))
```

```
;(add4args "F" "P" 2 "-") produces (FP2-)
(defun add4args (a b c d)
  (list(intern(concatenate 'string a b (princ-to-string c) d))))
```

```
;(add5args "S" "P" 3 "RHS" "-") produces (SP3RHS-)
(defun add5args (a b c d e)
  (list(intern(concatenate 'string a b (princ-to-string c) d e))))
```

```
(defun put-together(first second)
  (cond((null first) second)
    ((null second) first)
    ((and (atom (car first))
          (atom (car second))) (list first second))
    ((atom (car second))(append first (list second)))
    ((and (listp(car first))
          (listp(cadr second)))
      (put-together (cadr first) (put-together (car first) second)))
    (t(cons first second))))
```

```
;(defun de-string (x)
; (mapcar #(lambda (y) (intern y)) x))
```

Appendix B2

Listing of program implementing expansion theorem

```
;;;tempexpansion.lsp
```

```
;;;fully-expand
```

```
; A behaviour-expression is a list of atomic actions
; in sequence or with brackets.
; OR is prefix notation, all machines are bracketed
; e.g. (a b c) (a (+ (b) (c))) (a (+ (b c d) (e f g)))
; A composition is denoted by the symbol *, used as a prefix
; e.g. (* (a b c) (d (+ (e) (f))) (g) .....

; complementary terms are denoted by a trailing -, e.g. freda freda-

; Keep going 'till there are no * left

(defun fully-expand (expression restrictions)
  (cond ((not (findstars expression)) expression)
        (t (fully-expand (expand-terms expression restrictions)restrictions))))

(defun loud-fully-expand (expression restrictions)
  (cond ((not (findstars expression)) expression)
        (t (pprint expression)
            (loud-fully-expand (expand-terms expression restrictions)restrictions))))

(defun findstars (tree)
  (cond ((null tree) nil)
        ((equal tree **)t)
        ((atom tree) nil)
        (t (or (findstars (car tree))
                (findstars (cdr tree)) ) ) ))

;EXPAND
; An expand consists of an extract plus a silent-transitions.
; This takes an expression apart and puts it together again,
; but each composition will
; be replaced by an expansion (one-step) of the composition
(defun expand-terms (expression restrictions)
  (cond ((null expression) nil)
        ((atom expression) expression)
        ((equal (car expression) **)
         (combine-extract-and-silent (extract (cdr expression) restrictions)
                                     (silent-transitions (cdr expression) restrictions)))
        ((equal (car expression) '+) (let((result (combine-this-expand-and-rest
                                                    (expand-terms (cadr expression) restrictions)
                                                    (expand-terms (caddr expression) restrictions))))
                                     (cond((equal (length result) 1) result)
                                           ((AND(EQUAL (LENGTH RESULT) 2)
                                                (ATOM (CAR RESULT))) RESULT)
                                           (t(cons '+ result))))))
        (t (combine-this-expand-and-rest ;could save by checking for car = +
            (expand-terms (car expression) restrictions)
            (expand-terms (cdr expression) restrictions))))))
```

```
; EXTRACT
; This should return a list of 0 or more terms to be further expanded.
```

```
(defun extract (machines restrictions)
  (shuffle-extract nil machines restrictions))

(defun shuffle-extract (left right restrictions)
  (let((no-nil-right (remove nil right)))
    (cond ((null no-nil-right) nil)
          ((null (car no-nil-right))
           (shuffle-extract left (cdr no-nil-right) restrictions))
          ((equal (caar no-nil-right) '+) (combine-or-extract-and-rest
                                           (do-each-extract-of-or left (cdar no-nil-right)
                                                                (cdr no-nil-right) restrictions)
                                           (shuffle-extract (reverse (cons(car no-nil-right)
                                                                (reverse left)))
                                                                (cdr no-nil-right) restrictions)))
          ((member (caar no-nil-right) restrictions .test #'equalp)
           (shuffle-extract (cons (car no-nil-right) left)
                            (cdr no-nil-right)
                            restrictions))
          ((null (cdr no-nil-right))
           (list (compose-append1 (caar no-nil-right)
                                   (append left (cdar no-nil-right)
                                           (cdr no-nil-right)
                                           restrictions)))
                 (t (combine-this-extract-and-rest
                    (compose-append1 (caar no-nil-right)
                                      (append left (cdar no-nil-right)
                                              (cdr no-nil-right)
                                              restrictions)
                                      (shuffle-extract (reverse (cons
                                                                (car no-nil-right)
                                                                (reverse left)))
                                                                (cdr no-nil-right)
                                                                restrictions)))))))))
```

```
; This function takes a second argument which is a list of
;machines which are the body
; to the right, and to last is the restrictions.
;For each machine in the or, we want to
; produce a list of the compose, and link these with an OR. For example, (* ;(a) (+ (b)(c)) (d))
; would result in two expressions, vis:
; (b (* (a) (d))) and (c (* (a)(d)))
; so input: ((a)) ((b)(c)) (d) nil
; output: (+ (b (* (a) (d))) (c (* (a)(d))))
; another example (* (a) (+ (b (c)) (d)))
; input: ((a)) ((b (c)) (d)) nil nil
; output: (+ (b (* (a)(c)) (d (* (a) nil)))
; So for each machine, if it is linear we divide it into
; head and rest, and mark a
; compose of left rest and right, linked by a compose and prefixed by head.
; IF the machine isn't linear, we produce a set of terms recursively.
```

```
(defun do-each-extract-of-or (left machines right restrictions)
  (cond ((null machines) nil)
        (t
         (combine-this-or-and-rest
          (extract-one left (car machines) right restrictions)
          (do-each-extract-of-or left (cdr machines) right restrictions))))))
```



```

(remove (car match) (cdr right))))
  restrictions)
(shuffle-silent-transitions
(reverse(cons
(car right)
(reverse left)))
(cdr right)restrictions))))))

(defun compose-of (a b)
  (cond ((null a)b)
        ((null b)a)
        ((atom (car a)) (list a b))
        (t (append a b))))

(defun combine-this-silent-transition-and-next (a b)
  (cond ((null b) a)
        (t (join-in-order a b))))

; Do each thing in an or
(defun do-each-expand-of-or (machines restrictions)
  (cond ((null machines) nil)
        (t (combine-this-expand-or-and-rest
            (expand-terms (car machines) restrictions)
            (do-each-expand-of-or (cdr machines) restrictions) ) ) )

(defun do-each-silent-transition-of-or (left machines right restrictions)
  (cond ((null machines) nil)
        (t (combine-this-expand-or-and-rest
            (do-one-silent-transition-of-or left (cons (car machines) right) restrictions)
            (do-each-silent-transition-of-or left (cdr machines) right
            restrictions))))))

(defun do-one-silent-transition-of-or (left right restrictions)
  (cond ((null right) nil)
        ((< (length right) 2) nil)
        ((null (car right)) (do-one-silent-transition-of-or left (cdr right)
            restrictions))
        (t (let ((match (member (caar right) (cdr right)
            :test #'or-conamep))) ; Must be linear
            (cond ((null match) nil)
                  (t (combine-this-silent-transition-and-next
                    (compose-append ; delta
                    (concatenate 'string "~" (princ-to-string (caar right)))
                    (compose-of left (cons (cadar right)
                    (cons (cond ((equalp (caar match) '+)
                    (cadar(member(caar right) (cdr match)
                    :test #'conamep)))
                    (t (cadar match)))
                    (remove (car match) (cdr right)))))) restrictions)
            (do-one-silent-transition-of-or (reverse(cons (car right)
            (reverse left)))
            (cdr right) restrictions)))))))))

; This handles or's as well
(defun or-conamep (action1 machine)
  (cond ((equalp (car machine) '+)
        (sub-or-conamep action1 (cdr machine)))
        (t(conamep action1 machine) ) )

```

```
(defun sub-or-conamep (action machines)
  (cond ((null machines) nil)
        ((conamep action (car machines))
         (car machines))
        (t (sub-or-conamep action (cdr machines))))))
```

```
; Expects a linear machine and an action
(defun conamep (action1 machine)
  (let ((a1 (princ-to-string action1))      ;; turns atom to string
        (a2 (princ-to-string (car machine))))
    (and (equal (abs (- (length a1) (length a2))) 1)
          ;; checks diff length and equality
          (string-equal (string-right-trim "-" a1)
                        (string-right-trim "-" a2) ))))
```

```
;
; This is top level combination of extract and silent terms. e.g
; a|a- -> a(a-|nil) + (a-(a|nil)) + delta
; Two arguments are extract terms and silent terms. These are lists
; of machines. Possibilities are
; 1) No extract terms
; 2) No silent terms
; 3) One extract term
; 4) One silent term
; 5) OR of extract terms
; 6) OR of silent terms
```

```
(defun combine-extract-and-silent (extract silent)
  (cond ((equal silent nil)
        (cond ((equal (length (car extract)) 1) (car extract))
              ((> (length (car extract)) 1) (cons '+ extract))
              (t nil))))
        ((equal extract nil)
         (cond((and(< (length silent) 3)
                  (atom(car silent))) silent)
              ((equal (car silent) '+) silent)
              (t (cons '+ silent))))
        ((and (< (length extract) 3)
              (atom (car extract)))
         (cond((and(< (length silent) 3)
                  (atom (car silent)))
              (cons '+ (append (list extract) (list silent))))
              (t (cons '+ (append (list extract) silent))))))
        (t (cond((and(< (length silent) 3)
                  (atom (car silent)))
              (cons '+ (append extract (list silent))))
              (t( cons '+ (append extract silent))))))))
```

```
; Arguments are two results of calling expand-terms.
; Either may be nil
; Either may start with a +
; Or either may be a single expansion (i.e. a list starting without +)
```

```
(defun combine-this-expand-and-rest (first second)
  (cond ((and (null first)(null second)) nil)
        ((and (atom first) (null second)) (list first))
        ((null first) second)
        ((null second) first)
        ((atom first) (list first second))
```


; Given one machine, take it apart and eliminate any occurrences of restrictions

```
(defun derestrict-one-machine (machine restrictions)
  (let ((answer (derestrict-machine machine restrictions)))
    (cond((and(equal '+ (car answer))
              (equal (length (cdr answer)) 1))(cadr answer))
          ((and(equal '+ (car answer))
              (null (cdr answer)))nil)
          (t answer))))

(defun derestrict-machine
  (machine restrictions)
  (cond ((null machine) nil)
        ((and (atom machine)
              (member machine restrictions :test #'equalp))
         nil)
        ((atom machine) machine)
        ((member (car machine) restrictions :test #'equalp) nil)
        (t
         (null-check-cons (derestrict-machine (car machine) restrictions)
                          (derestrict-machine (cdr machine) restrictions))))))

(defun null-check-cons (first second)
  (cond ((null first)
         second)
        ((and (equal '+ first)
              (null second)) nil)
        (t
         (cons first second))))

(defun null-check-list (first second)
  (cond ((null second)
         (list first))
        ((null first) second)
        (t
         (list first second))))

(defun clever-append (first second)
  (cond
   ((and (null first)
         (null second)) nil)
   ((null first) second)
   ((null second) first)
   ((and (equal (car first) '+)
         (equal (car second) '+))
    (append first (cdr second)))
   ((equal (car first) '+)
    (append first second))
   ((equal (car second) '+)
    (cons '+ (append first (cdr second))))
   (t(cons '+ (join-in-order first second))) ))

(defun shuffle-extract-append (first second)
  (cond ((null second) (cons '+ first))
        ((and (null second)
              (equal (length first) 1))
         ;Only happens with restrictions
         (car first))
        ((equal (car second) '+)
         (cons '+ (append first (cdr second))))
        (t(cons '+ (append first second))) ))
```

```
(defun silent-append (first second)
  (cond ;((null first) second)
        ((null second) first)
        ((and (null second)
              (equal (length first) 1))
         (car first))
        ((and (equal (car first) '+)
              (equal (car second) '+))
         (append first (cdr second)))
        ((equal (car first) '+) (append first second))
        ((equal (car second) '+) (cons '+ (append first (cdr second))))
        (t (cons '+ (append first second)))))

(defun clever-cons (a b)
  (cond ((null b) a)
        (t (cons a b))))
```

Appendix B3

Listings from production rule system

;;; program initialising working memory

; Initwm.lsp

;;; go through ccstree and identify choicepoints
 ;;; type and level-number, starting at level 1
 ;;; first version for outputting on screen in legible format

```
;(defun initwm(ccstree)
; (tidy-up (cons '((position 1)) (direction forward))
; (categorise ccstree '(1))))
```

```
;(defun tidy-up (alist)
; (cond((null alist)nil)
; (t (terpri)(sortout (car alist))
; (tidy-up (cdr alist)))))
```

```
;(defun sortout (alist)
; (cond((null alist)nil)
; (t(print (car alist))
; (sortout (cdr alist)))))
```

```
(defun initwm(ccstree)
(cons '((position 1))(direction forward)) ;head of working memory
(categorise ccstree '(1)))
```

;;; look at each node in turn and see if it is a choicepoint
 ;;; pick out the conjunctive and disjunctive +'s
 ;;; and the fail nodes

```
(defun categorise(alist choicepointnum )
(cond((null alist) nil)
((final (car alist))(recordchoicepoint
(car alist) choicepointnum))
((equalp '+ (caadr alist)) ;'choice node is followed by '+'
(joinon (recordchoicepoint
(car alist) ;record type of choice node
choicepointnum)
(categorise (cadr alist) ;carry on down ccstree, level deeper
(next-level choicepointnum))))
((equalp '+ (car alist)) ;at '+'
(takeouttail (cdr alist) ;go down the choices in tail
choicepointnum))
((equalp "~F" (firstchar (car alist))) ;a failure means choosing
(recordchoicepoint
(car alist)
choicepointnum ))
(t (joinon (recordnode (car alist) choicepointnum) ;common or garden nodes
(categorise (cadr alist)
(next-level choicepointnum))))) ;record and carry on
```

```
(defun recordnode(item choicepointnum )
(list(list 'choicepoint choicepointnum )
(list 'type choicepointnum 'nochoice-node) ;dont involve a choice
(list 'name choicepointnum item)))
```

```
(defun next-level(levelnum) ;level marker
(reverse (cons 1 ;adds 1 to existing level
(reverse levelnum)))) ;e.g. (1 1) to (1 1 1)
```

;; when dealing with a choice inside a tail, take first element, look at it,
 ;; then go on down list, at top level stay same choicepoint number

```
(defun takeouttail(tail choicepointnum )
  (cond((null tail) nil) ;look at topmost choice
        (t(joinon (categorise (car tail)
                              choicepointnum)
                  (takeouttail(cdr tail) ;then take the other(s)
                              (addit choicepointnum) )))))
```

```
(defun addit(choicepointnum) ;each clause of disjunct
  (reverse ;is sub-numbered in order
    (cons ;e.g.(1 2 1) (1 2 2)(1 2 3)..
      (+ (car (reverse choicepointnum))1)
      (cdr (reverse choicepointnum))))))
```

```
(defun final(item)
  (and(equalp "S"(subseq(princ-to-string item) 0 1))
    (equalp "-"(lastchar (princ-to-string item)))))
```

;; label choicepoints according to sort: 'conjunctive, 'disjunctive
 ;; 'conjunctive prime or a 'no choice' node
 ;; probably need changing later

```
(defun recordchoicepoint(item choicepointnum)
  (cond ((and(equalp "S"(subseq(princ-to-string item) 0 1))
             (equalp "-"(lastchar (princ-to-string item))))
        (record-final-success item choicepointnum))
        ((equalp "S" (subseq (princ-to-string item) 0 1))
         (record-disjunct item choicepointnum))
        ((rside item) (record-disjunctrhs item choicepointnum ))
        ((equalp "~F"(firstchar item))
         (record-failchoice item choicepointnum ))
        ((equalp "-" (lastchar item))
         (record-disjunct item choicepointnum ))
        (t (record-disjunctprime
            item choicepointnum ))))
```

;;putting the appropriate tags on the choicepoint

```
(defun record-final-success(item choicepointnum )
  (list(list 'choicepoint choicepointnum )
        (list 'type choicepointnum 'final-success)
        (list 'name choicepointnum item)))
```

```
(defun record-disjunct(item choicepointnum )
  (list(list 'choicepoint choicepointnum )
        (list 'type choicepointnum 'disjunct)
        (list 'name choicepointnum item)))
```

```
(defun record-disjunctrhs(item choicepointnum )
  (list(list 'choicepoint choicepointnum )
        (list 'type choicepointnum 'disjunctrhs)
        (list 'name choicepointnum item)))
```

```
(defun record-disjunctprime(item choicepointnum )
  (list(list 'choicepoint choicepointnum )
        (list 'type choicepointnum 'disjunctprime)
        (list 'name choicepointnum item)))
```

```
(defun record-failchoice(item choicepointnum )
```

```

(list(list 'choicepoint choicepointnum )
      (list 'type choicepointnum 'failchoice)
      (list 'name choicepointnum item)))

;;identifier bits
(defun rside(item)
  (and(> (length item) 4)
       (equalp "RHS-"
               (string-trim
                (subseq item 0 (- (length item) 4)) item))))

(defun lastchar (item)
  (string-trim
   (subseq item 0 (- (length item) 1)) item))

(defun firstchar(item)
  (subseq (princ-to-string item) 0 2))

;;glue bits

(defun joinon(a b)
  (cond((null a) b)
        ((null b) a)
        ((and(atom (caar a)
                  (atom (caar b))))
         (cons a (list b)))
        ((and(listp (caar a)
                    (listp (caar b)))
         (append a b))
        ((listp (caar a))(append a (list b)))
        (t (cons a b))))

;;ruleset for normal Prolog search

(((position ?node)(direction forward)(type ?node nochoice-node))
 ((position (next-level ?node ))(direction forward)))

(((position ?node)(direction backward)(type ?node nochoice-node))
 ((position (stepback ?node ))(direction backward)))

(((position ?node)(direction forward)(type ?node disjunctrhs))
 ((position (next-level ?node))(direction forward)))

(((position ?node)(direction backward)(type ?node disjunctrhs))
 ((position (stepback ?node))(direction backward)))

(((position ?node)(direction forward)(type ?node disjunct))
 ((position (next-level ?node))(direction forward)))

(((position (1))(direction backward)(type ?node disjunct)
 ((not(up-levels(next-level ?node))))
 ((halt))))

(((position ?node)(direction backward)(type ?node disjunct)
 ((up-levels (next-level ?node))))
 ((position (up-levels(next-level ?node)))(direction forward)))

(((position ?node)(direction backward)(type ?node disjunct)
 ((not(up-levels (next-level ?node))))
 ((position(stepback ?node))(direction backward)))

```



```

(((position ?node)(direction forward)(type ?node disjunctprime))
  ((position (next-level ?node ))(direction forward)))

(((position ?node)(direction backward)(type ?node disjunctprime))
  ((position (stepback ?node ))(direction backward)))

(((position ?node)(direction forward)(type ?node failchoice))
  ((position (stepback ?node))(direction backward)))

(((position ?node)(direction forward)(type ?node final-success))
  ((halt))))

;;search cycle

;;; system cycles through rules to find next node
;;; in path being followed through ccstree

(defun beginsearch(ccstree rules)
  (asearch(initwm ccstree)rules) ;label nodes of ccstree <initwm.lsp>

(defun asearch(wm rules)(print 'car-of-wm=)(print (car wm))
  (cond((equalp '(HALT))(car wm))
    (reverse(path (pathlist wm) ;return the path taken
      (wmchoicepoints wm))))
  (t(asearch (choice wm rules)
    rules))))

;;; choice uses most recent element in wmem,
;;; finds what kind of node is identified
;;; matches it to lhs of rule and fires matching rhs of rule

;(varvalue returns instantiated lhs of rule)
;(findrhs returns the rhs of that rule)

(defun choice (wm rules)
  (let((varvalue (matches (caar rules) wm))) ;take first rule
    (cond((null varvalue) ;if not a match try next rule
      (choice wm (cdr rules)))
      (t (execute varvalue ;if there is a match
        (findrhs varvalue rules) wm)))) ;instantiate and do rhs of rule

;;; fn matches returns a lhs of a rule that has been
;;; instantiated to match the most recent node
;;; position and type in wmem

(defun matches (lhsrule wm)
  (and(equalp (cadr lhsrule) ;check the direction of rule
    (cadr wm)) ;match in wmem
    (findrule (subst (cadaar wm) ;substitute position-id
      "?node" ;find node-id match in w-mem
      lhsrule :test #'equalp)
      wm wm)))

(defun findrule(instantiated-lhs wm wm1)
  (cond ((null wm) nil) ;matches a choicepoint if
    ((and (equalp (rulepos instantiated-lhs) ;node-id rule equal to
      (cadr(cadr wm))) ;node-id choicepoint
      (equalp (ruletype instantiated-lhs) ;and type-id of rule equalto
        (wmtype wm)) ;type-id of choicepoint

```

```

(or(null (caddr instantiated-lhs))
 (satisfied (caddr instantiated-lhs) wm1 )))and constraints are met
instantiated-lhs)
(t (findrule
instantiated-lhs
(cdr wm) wm1)))) ;otherwise look in tail of w-mem

;;; fn satisfies checks constraints on lhs of rule
;;; taking each constraint in turn

(defun satisfied(condbit wm)
(cond((null condbit) t)
(t (and(satisfy (car condbit)wm ) ;satisfy first constraint
(satisfied (cdr condbit) wm )))) ;and the rest

(defun satisfy (condcar wm )
(eval (addargs condcar wm )))

(defun addargs (condcar wm )
(cond ((null condcar) nil)
((or (equal (car condcar) 'not)
(equal (car condcar) 'next-level)
(equal (car condcar) 'stepback)
(equal (car condcar) 'following))
(list (car condcar)
(addargs (cadr condcar) wm )))
((equal (car condcar) 'used)
(list 'used (addargs (cadr condcar) wm )
(list 'quote (pathlist wm))))
((equal (car condcar) 'up-levels)
(list 'up-levels (addargs (cadr condcar) wm )
(list 'quote (pathlist wm))
(list 'quote (working-mem wm) )))
((equal(car condcar) 'previously-used-disjunct)
(list 'previously-used-disjunct
(addargs(cadr condcar) wm)
(list 'quote (pathlist wm))))
((equal(car condcar) 'rule-clause)
(list 'rule-clause
(addargs(cadr condcar)wm)
(list 'quote (wmchoicepoints wm))))
((equal (car condcar) 'exists-lastdisjunctrhs)
(list 'exists-lastdisjunctrhs
(addargs(cadr condcar) wm) (list 'quote wm)))
(t (list 'quote condcar))))

;;; findrhs checks out the rhs of
;;; instantiated lhs of rule

(defun findrhs(lhs rules)
(cond((null rules)nil) ;if position values match head
((same lhs (caar rules)) ;of rules and righthand side is
(cond((equalp '((halt))(cadar rules)) ;'halt' return that
'((halt))) ;otherwise substitute in
(t(subst (rulepos lhs) ;values of position to
(cadadr(car(cadar rules))) ;lefthand side and
(cadar rules)))))) ;return matching righthand side
(t (findrhs lhs (cdr rules)))) ;if no match, try rest of rules

;;; match if the direction and node type
;;; are the sameand the constraints are the same

```

```

(defun same(lhs toprule)
  (and(equalp (ruledirection lhs)           ;compare direction
             (ruledirection toprule))
        (equalp (ruletype lhs)             ;compare node-type
                 (ruletype toprule))
        (same-constraints (caddr lhs) (caddr toprule)))) ;compare constraints

(defun same-constraints(constraints-lhs constraints-toprule)
  (cond((and(null constraints-lhs)
             (null constraints-toprule)) t) ;compare list of constraints

        ((and(lookat (car constraints-lhs)(car constraints-toprule))
              (same-constraints
               (cdr constraints-lhs)(cdr constraints-toprule))))))

(defun lookat(alist alist1) ;compare each constraint
  (equalp (car alist)(car alist1)))

;;; having found the matching lhs and rhs
;;; execute fires righthand side of rule

(defun execute(lhs rhs wm)
  (cond((equalp '((halt)) rhs)
        (cons '((halt)) wm) ;if 'halt is rhs, output wmem
        ((equalp 'next-level (rhs-action rhs)) ;nextlevel goes along the branches
         (nextlevel lhs rhs wm))
        ((equalp 'stepback (rhs-action rhs)) ;step back node by node
         (goback lhs rhs wm))
        ((equalp 'last-disjunctrhs (rhs-action rhs))
         (lastdisjunctrhs rhs wm))
        ((equalp 'up-level (rhs-action rhs)) ;up-level(s) ;uplevel goes up branches
         (uplevel lhs rhs wm))))

;functions used for constructing output list
;showing path taken through CCStree

;list1 is list of positions from wmem
;list2 is list of choicepoints from wmem

(defun path(list1 list2) ;list of positions
  (cond((null list1) nil) ;and list of node names
        (t(joint(tie (car list1) list2) ;put in a list as identified
                  (path(cdr list1) list2))))))

(defun joint(a b) ;with a line break,
  (append a b)) ;more legible as output

;tie in position to corresponding node-name

(defun tie (item alist)
  (cond((equalp '((halt)) item) ;not a position
        nil)
        ((null alist) nil)
        ((equalp (rulepos item) (wmpos (car alist))) ;if position matches
         (caddr(caddr (car alist)))) ;top choicepoint, return name,
        (t (tie item (cdr alist)))) ;otherwise go on down tail

;(defun tie (item alist)
; (cond((equalp '((halt)) item) ;not a position

```

```

:      nil)
:      ((null alist) nil)
:      ((and (equalp (rulepos item) (wmpos (car alist))) ;if position matches top choicepoint
:            (equalp (choicepointtype (car alist)) 'disjunct));and is a disjunct
:            (caddr(caddr (car alist))))
:      (t (tie item (cdr alist))))

```

;a function that only allows disjuncts in a forward direction
;and disjuncts in backwards when changed to 'r' version
;plus sp and sp- or fp-

;reduce wmem to choicepoints only,

```

(defun wmchoicepoints (wm) ;list of 'choicepoints' in working memory
  (cond((null wm) nil)
        ((equalp 'position (caar (car wm))) ;ignore positions
         (wmchoicepoints(cdr wm)))
        (t (joinup (car wm) ;add choicepoints to list
                    (wmchoicepoints(cdr wm))))))

```

;;;returns list of node-positions from wmem

```

(defun pathlist(list1)
  (cond((equalp (caaar list1) 'choicepoint) nil) ;dont include original
        (t(cons (car list1) ;choicepoints, only positions
                 (pathlist(cdr list1))))))

```

;glue for list of choicepoints

```

(defun joinup(a b)
  (cond((and(null a)(null b))nil)
        ((null a)b)
        ((null b)a)
        ((and(atom(caar a)
                  (atom (caar b)))(cons a (list b)))
         (atom (caar a))(cons a b))
        ((atom (caar b))(append a (list b)))
        (t (cons a b))))

```

;;; functions used as constraints in lhs of rules at disjuncts

;;; for facts before rules
;;; need a check to find if there is a fact
;;; which could be used instead of a rule

;fn 'rule-clause' uses results of (following node)
;and choicepoints stored in working memory (wmchoicepoints wm)

```

(defun rule-clause (position alist)
  (rhside (car(tie-up position alist))))

```

```

(defun tie-up (item alist)
  (cond((null alist) nil)
        ((equalp item (wmpos (car alist)))
         (caddr(caddr(car alist)))) ;if a choicepoint return it
        (t (tie-up item (cdr alist)))) ;otherwise go on to tail

```

;;;for onepointer per clause
;;;a check to see if node has been used

;fn uses 'next-level' position of node
;and the pathlist of nodes used

```
(defun previously-used-disjunct(position alist)
  (cond((equalp '(1) position) nil) ;look at last position
        ((equalp 1 (car(reverse position))) ;if the level is '1'
         (previously-used-disjunct(stepback position) alist));check further back,
        ((is-lower position alist) ;used previously if level
         (t (previously-used-disjunct ;is higher than '1'
              (stepback position) alist)))) ;or check further back
```

```
(defun is-lower(position alist) ;checks if a lower node exists
  (cond((null alist) nil) ;by looking one level lower
        ((equalp (lower-branch position) (caddr alist)) ;yes, if lower level is head
         (t (is-lower position (cdr alist)))) ;or tail of pathlist
```

```
(defun lower-branch (position)
  (reverse(cons (- (car (reverse position))1) ;going 'down' one level
                (cdr (reverse position)))) ;e.g. (1 1 2) to (1 1 1)
```

;;;for redo from left model
;;;checks on leftmost goal rather than
;;;most recent subgoal

```
(defun exists-lastdisjunctrhs(position wm)
  (cond((null wm) nil)
        ((and (equalp (caddr(cadr (car wm))) ;if there is a node type
                    'disjunctrhs) ;which is a rhs-disjunct
              (> (length position) ;and it is further back
                  (length(cadadr(car wm)))) ;in path of nodes used
              (cadadr (car wm)))) ;return that position
        (t(exists-lastdisjunctrhs position (cdr wm)))))
```

;;;fn.s used by 'execute' when firing righthand side
;;;of rule, to update the position of node

```
(defun lastdisjunctrhs(rhs wm)
  (cons(list
        (cons(caar rhs) ;"position"
              (list(next-level (exists-lastdisjunctrhs ;find leftmost goal
                              (cadadr(car rhs)) wm)))) ;add node number
        (cadr rhs) wm)) ;add node direction
```

```
(defun nextlevel(lhs rhs wm)
  (cons
   (list(cons (caar rhs) ;"position"
              (list
               (next-level(cadar lhs)))) ;add node number
         (cadr rhs) wm)) ;add node direction
```

```
(defun uplevel(lhs rhs wm)
  (cons(list
        (cons(caar rhs)(list ;adding a new position to pathlist
                (up-levels(next-level(cadar lhs)) ;when it is one level up
                            (pathlist wm) ;rather than along the tree
                            (working-mem wm))))
        (cadr rhs)
        wm))
```

```

:(defun goback(lhs rhs wm)
: (let((step (stepback-to-disjunct(cadar lhs)      ;when updating position
:           (wmchoicepoints wm)))) ;is stepping back,
:   (cond((null step)(cons '((halt)) wm)) ;and step back is last, halt
:         (t (cons(list(cons(caar rhs)           ;otherwise list "position"
:                           (list step))       ;node number and direction
:                           (cadr rhs)) wm))))))
(defun goback(lhs rhs wm)
(let((step (stepback(cadar lhs)))) ;removes one level going back
(cond((null step) (cons '((halt)) wm)) ;if result is last step add halt
      (t(cons(list(cons(caar rhs)           ; otherwise put new pos. in wm
:                 (list step))
:                 (cadr rhs))wm))))))

(defun stepback(nodeposition)
(reverse ;removes one level going back
(cdr (reverse nodeposition))) ;along the tree

:(defun stepback-to-disjunct(position alist)      ;find last disjunct
: (cond((null position) nil)
:       ((disjunct-found (stepback position) alist) ;return it if found
:         (stepback position))
:       (t(stepback-to-disjunct (stepback position) alist))) ;otherwise go back a step

:(defun disjunct-found (position alist)
: (cond((null alist) nil)
:       ((and(equalp position ;if the position number is the same
:             (wmpos (car alist)) ;and the type is 'disjunct'
:             (equalp (choicepointtype (car alist)) ;eureka
:                     'disjunct)))
:       (t(disjunct-found position (cdr alist)))) ;otherwise look on down list

:(defun next-level(levelnum)
: (reverse (cons 1 ;adds one level
:              (reverse levelnum))) ;going along tree

(defun following(position) ;finds next but one level
(next-level(next-level position)) ;along the tree

(defun up-levels(choicepointnum wm wm1) ;going up a branch level
(let((uplevelnode (up-level choicepointnum)) ;as opposed to along
      (cond ((not(wm-mem uplevelnode wm1)) nil) ;returns nil if no higher node
            ((used uplevelnode wm)(up-levels uplevelnode wm wm1));tries higher if used
            (t uplevelnode)))) ;or returns higher branch

(defun up-level(choicepointnum)
(cond((equalp 1 (length choicepointnum)) ;if down to last node of pathlist,
      (reverse(cons ;a bit different,
:               (+ (car choicepointnum)1) ;no list to reverse
:               choicepointnum)))
      (t (reverse ;normally, adds a level going up
:               (cons ;the branches (1 1) to (1 2)
:                 (+ (car (reverse choicepointnum))1)
:                 (cdr (reverse choicepointnum)))))))

(defun wm-mem(position alist) ;check on existence of a node
(cond((null alist) nil)
      ((equalp position (cadar alist)) t) ;exists if a member
      (t (wm-mem position (cdr alist)))) ;of original working memory

(defun working-mem(wm)

```

```

(cond((null wm)nil) ;original working memory
      ((equalp (caaar wm) 'position) ;records choicepoints
        (working-mem(cdr wm))) ;generated for program concerned
      (t (cons(caaar wm)
              (working-mem(cdr wm))))))

(defun used (anode wm) ;check to see if node has been used
  (or(null anode) ;disjunct node has been used if
      (pathlistmem ;it is a member of pathlist
        anode ;(or it is nil - thanks a million mark)
        wm)))

(defun pathlistmem (anodepos wm )
  (cond((null wm ) nil)
        ((inwmem anodepos (car wm )) t) ;if is in the head of working memory
        (t(pathlistmem anodepos (cdr wm )))) ;or in the tail
  (defun inwmem(anode carwm )
    (equalp anode (cadar carwm ))) ;there you go

;;; functions for pulling out certain bits of rule
;;; and for identifying node position and type etc.

(defun ruletype (rule) ;rule = (caar rules)
  (caddr(caddr rule)))

(defun rulepos(rule)
  (cadar rule))

;;;same again for identifying node position
;;;and type in wmem

(defun wmtypewm(wm)
  (caddr(cadr(cadr wm))))

(defun wmposwm(wm)
  (cadadr wm))

(defun ruledirection(rule)
  (cadr rule))

(defun wmdirection(wm)
  (cadar wm))

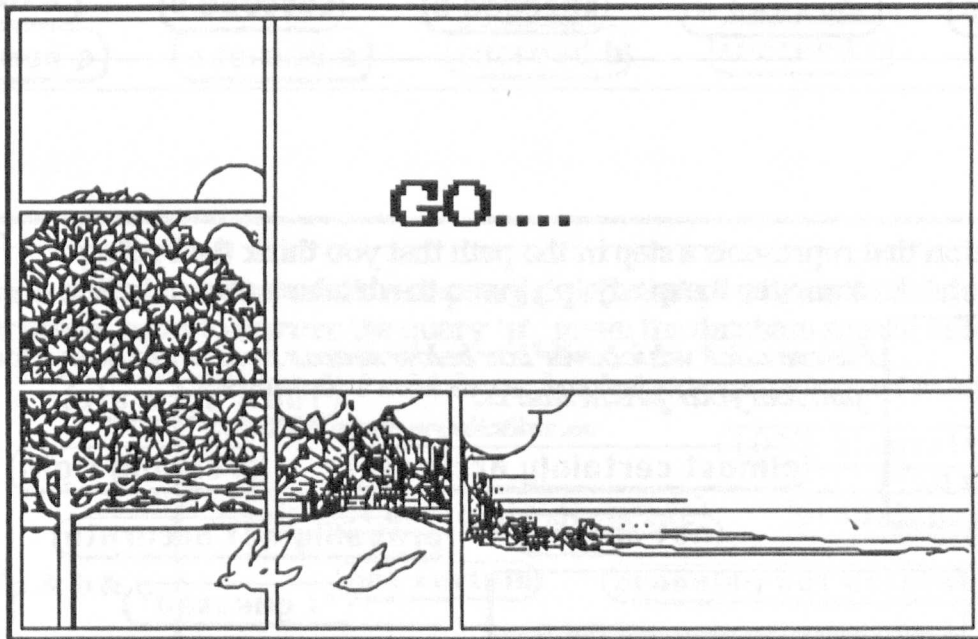
(defun rhs-action(rhs)
  (caadr(car rhs)))

(defun choicepointtype(alist) ;e.g. 'disjunct' or 'conjunct'
  (caddr(cdr alist)))

```

Appendix C1

Screen dumps of interface and programs used in experiment

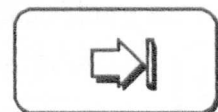
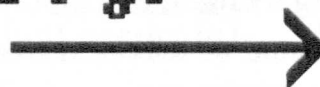


Hi - thank you for coming !!

Thank you for taking part in this experiment. Its purpose is to help find out which aspects of Frulog are difficult for novices. This exercise looks in particular at how people imagine the Frulog interpreter finds an answer to a query; that is, what steps the Frulog interpreter takes in proving a goal. Take your time over the example given on the next screen and then go on to the rest of the programs.


Don't spend too long over the programs you will be given, I am more interested in which steps you predict the interpreter will take than in whether they are the "right" or "wrong" steps. After you have finished, I'll be glad to answer any queries you have about the programs.

Off we go



try p	try a	try b	try c	Example
fail p	fail a	fail b	fail c	
succeed p	succeed a	succeed b	succeed c	

Click each button that represents a step in the path that you think the Prolog interpreter takes to answer the query 'p', given the database shown below



b.
a.
p if a & b & c.

Please click whichever box below seems nearest to how accurate you feel your prediction is:

100% accurate

probably right

almost certainly accurate

probably not accurate

not certain

"I guessed"

←

→

If you find the notation a little strange, read the following section, otherwise go straight on to the rest of the programs by clicking on the "Let's go" button below.

It may help to compare these two programs, the first is taken from some of the examples in your "Artificial Intelligence" handbook for D089. They are basically the same programs, i.e. something is true if three things about it can be proved true, in the first it is three facts about an aeroplane, in the second, three facts 'a' 'b' 'c':

<pre> departs(BA 137 1040). origin(BA1f37 Chicago). likes(Matthew BA137) if origin(BA 137 Chicago) & departs(BA137 1040) & arrives(BA137 1250). </pre>	b. a. p if a & b & c.
---	-----------------------------

In each of these programs the Prolog interpreter would take exactly the same steps, in each case finding a match in a conditional logical sentence, then trying

←

to prove each subgoal of that sentence in turn.

Let's go

problem one

<input type="button" value="try p"/>	<input type="button" value="try a"/>	<input type="button" value="try b"/>	<input type="button" value="try c"/>
<input type="button" value="fail p"/>	<input type="button" value="fail a"/>	<input type="button" value="fail b"/>	<input type="button" value="fail c"/>
<input type="button" value="succeed p"/>	<input type="button" value="succeed a"/>	<input type="button" value="succeed b"/>	<input type="button" value="succeed c"/>

Click each button that represents a step in the path that you think the Prolog interpreter takes to answer the query 'p', given the database shown below

a.

b.

b.

p if a & b & c.

Please click whichever box below seems nearest to how accurate you feel your prediction is:

➔

problem two

<input type="button" value="try p"/>	<input type="button" value="try a"/>	<input type="button" value="try b"/>	<input type="button" value="try c"/>
<input type="button" value="fail p"/>	<input type="button" value="fail a"/>	<input type="button" value="fail b"/>	<input type="button" value="fail c"/>
<input type="button" value="succeed p"/>	<input type="button" value="succeed a"/>	<input type="button" value="succeed b"/>	<input type="button" value="succeed c"/>

Click each button that represents a step in the path that you think the Prolog interpreter takes to answer the query 'p', given the database shown below

a.

a.

b.

p if a & b & c.

Please click whichever box below seems nearest to how accurate you feel your prediction is:

➔

problem three

try p	try a	try b	try c
fail p	fail a	fail b	fail c
succeed p	succeed a	succeed b	succeed c

Click each button that represents a step in the path that you think the Prolog interpreter takes to answer the query 'p', given the database shown below

a.
b.
a.
p if a & b & c.

Please click whichever box below seems nearest to how accurate you feel your prediction is:

100% accurate

almost certainly accurate
probably right

not certain
probably not accurate

[]
"I guessed"

➔

problem four

try p	try a	try b	try c
fail p	fail a	fail b	fail c
succeed p	succeed a	succeed b	succeed c

Click each button that represents a step in the path that you think the Prolog interpreter takes to answer the query 'p', given the database shown below

p if a.
a.

Please click whichever box below seems nearest to how accurate you feel your prediction is:

100% accurate

almost certainly accurate
probably right

not certain
probably not accurate

[]
"I guessed"

➔

problem five


<input type="button" value="try p"/>	<input type="button" value="try a"/>	<input type="button" value="try b"/>	<input type="button" value="try c"/>	<input type="button" value="try x"/>
<input type="button" value="fail p"/>	<input type="button" value="fail a"/>	<input type="button" value="fail b"/>	<input type="button" value="fail c"/>	<input type="button" value="fail x"/>
<input type="button" value="succeed p"/>	<input type="button" value="succeed a"/>	<input type="button" value="succeed b"/>	<input type="button" value="succeed c"/>	<input type="button" value="succeed x"/>

Click each button that represents a step in the path that you think the Prolog interpreter takes to answer the query 'p', given the database shown below

p if a & b & c.
 a if x
 x
 b
 a

Please click whichever box below seems nearest to how accurate you feel your prediction is:

<input type="button" value="100% accurate"/>
<input type="button" value="almost certainly accurate"/> <input type="button" value="probably right"/>
<input type="button" value="not certain"/> <input type="button" value="probably not accurate"/>
<input type="button" value="I guessed"/>



problem six



<input type="button" value="try p"/>	<input type="button" value="try a"/>	<input type="button" value="try b"/>	<input type="button" value="try c"/>	<input type="button" value="try e"/>
<input type="button" value="fail p"/>	<input type="button" value="fail a"/>	<input type="button" value="fail b"/>	<input type="button" value="fail c"/>	<input type="button" value="fail e"/>
<input type="button" value="succeed p"/>	<input type="button" value="succeed a"/>	<input type="button" value="succeed b"/>	<input type="button" value="succeed c"/>	<input type="button" value="succeed e"/>

Click each button that represents a step in the path that you think the Prolog interpreter takes to answer the query 'p', given the database shown below

p if a & b & c.
 a.
 b.
 a if e.
 e.

Please click whichever box below seems nearest to how accurate you feel your prediction is:

<input type="button" value="100% accurate"/>
<input type="button" value="almost certainly accurate"/> <input type="button" value="probably right"/>
<input type="button" value="not certain"/> <input type="button" value="probably not accurate"/>
<input type="button" value="I guessed"/>

Thankyou for your co-operation, before you go,
click on the activity that seems most attractive at this moment

you are now free to:

- collapse into bed
- collapse at the bar
- lunch at the refectory
- lynch the experimenter

Appendix C2

Results of machine-analysis 1988 data

Interpretation of output

"match-found-to-PROB1TRYONCE"
indicates that on problem one (PROB1)
the student's prediction showed the typical
error associated with the 'try once and pass'
misconception.

"match-found-to-PROB5MULTIPLEF"
would indicate that on problem five the
student showed evidence of both the 'try
once and pass' error and the 'facts before
rules' error.

"match-found-to-PROB4NORM"
indicates that the student on problem four
the student gave a correct prediction.

subject no. 1

"match-found-to-PROB1TRYONCE"
"match-found-to-PROB2TRYONCE"
"match-found-to-PROB3TRYONCE"
"match-found-to-PROB4NORM"
nil
nil

subject no. 2

"match-found-to-PROB1TRYONCE"
"match-found-to-PROB2TRYONCE"
"match-found-to-PROB3TRYONCE"
"match-found-to-PROB4NORM"
nil
nil

subject no. 3

"match-found-to-PROB1TRYONCE"
"match-found-to-PROB2TRYONCE"
"match-found-to-PROB3TRYONCE"
"match-found-to-PROB4NORM"
nil
nil

subject no. 4

nil
nil
nil
"match-found-to-PROB4NORM"
nil
nil

subject no. 5

nil
"match-found-to-PROB2TRYONCE"
"match-found-to-PROB3TRYONCE"
"match-found-to-PROB4NORM"
"match-found-to-PROB5MULTIPLEF"
"match-found-to-PROB6TRYONCE"

subject no. 6

"match-found-to-PROB1TRYONCE"
"match-found-to-PROB2TRYONCE"
"match-found-to-PROB3TRYONCE"
"match-found-to-PROB4NORM"
"match-found-to-PROB5MULTIPLEF"
"match-found-to-PROB6TRYONCE"

subject no. 7

"match-found-to-PROB1TRYONCE"
"match-found-to-PROB2TRYONCE"
"match-found-to-PROB3TRYONCE"
"match-found-to-PROB4NORM"
nil
"match-found-to-PROB6TRYONCE"

subject no. 8

"match-found-to-PROB1TRYONCE"

"match-found-to-PROB2TRYONCE"

nil

"match-found-to-PROB4NORM"

"match-found-to-PROB5MULTIPLEF"

"match-found-to-PROB6TRYONCE"

subject no. 9

nil

nil

nil

nil

nil

nil

subject no. 10

"match-found-to-PROB1TRYONCE"

"match-found-to-PROB2TRYONCE"

"match-found-to-PROB3TRYONCE"

"match-found-to-PROB4NORM"

"match-found-to-PROB5MULTIPLEF"

"match-found-to-PROB6TRYONCE"

subject no. 11

"match-found-to-PROB1TRYONCE"

"match-found-to-PROB2TRYONCE"

"match-found-to-PROB3TRYONCE"

"match-found-to-PROB4NORM"

nil

nil

subject no. 12

"match-found-to-PROB1TRYONCE"

"match-found-to-PROB2TRYONCE"

"match-found-to-PROB3TRYONCE"

"match-found-to-PROB4NORM"

"match-found-to-PROB5TRYONCE"

"match-found-to-PROB6TRYONCE"

subject no. 13

nil

nil

nil

nil

nil

nil

subject no. 14

"match-found-to-PROB1TRYONCE"

"match-found-to-PROB2TRYONCE"

"match-found-to-PROB3TRYONCE"

"match-found-to-PROB4NORM"

"match-found-to-PROB5TRYONCE"

"match-found-to-PROB6TRYONCE"

subject no. 15

nil

nil

nil

nil

nil

nil

subject no. 16

nil

nil

nil

"match-found-to-PROB4NORM"

nil

nil

subject no. 17

"match-found-to-PROB1TRYONCE"

"match-found-to-PROB2TRYONCE"

"match-found-to-PROB3TRYONCE"

"match-found-to-PROB4NORM"

nil

"match-found-to-PROB6TRYONCE"

subject no. 18

"match-found-to-PROB1TRYONCE"

"match-found-to-PROB2TRYONCE"

"match-found-to-PROB3TRYONCE"

"match-found-to-PROB4NORM"
"match-found-to-PROB5TRYONCE"
"match-found-to-PROB6TRYONCE"

subject no. 19

nil
nil
nil
nil
nil
nil

subject no. 20

"match-found-to-PROB1TRYONCE"
"match-found-to-PROB2TRYONCE"
"match-found-to-PROB3TRYONCE"
"match-found-to-PROB4NORM"
"match-found-to-PROB5MULTIPLEF"
"match-found-to-PROB6TRYONCE"

subject no. 21

nil
nil
nil
"match-found-to-PROB4NORM"
nil
nil

subject no. 22

"match-found-to-PROB1TRYONCE"
"match-found-to-PROB2TRYONCE"
"match-found-to-PROB3TRYONCE"
"match-found-to-PROB4NORM"
"match-found-to-PROB5MULTIPLEF"
"match-found-to-PROB6TRYONCE"

subject no. 23

"match-found-to-PROB1TRYONCE"
"match-found-to-PROB2TRYONCE"
"match-found-to-PROB3TRYONCE"
"match-found-to-PROB4NORM"

"match-found-to-PROB5TRYONCE"
"match-found-to-PROB6TRYONCE"

subject no. 24

nil
nil
nil
nil
nil
nil

subject no. 25

"match-found-to-PROB1TRYONCE"
"match-found-to-PROB2TRYONCE"
"match-found-to-PROB3TRYONCE"
"match-found-to-PROB4NORM"
"match-found-to-PROB5MULTIPLEF"
"match-found-to-PROB6TRYONCE"

subject no. 26

nil
nil
nil
nil
nil
nil

subject no. 27

"match-found-to-PROB1TRYONCE"
"match-found-to-PROB2TRYONCE"
"match-found-to-PROB3TRYONCE"
"match-found-to-PROB4NORM"
"match-found-to-PROB5TRYONCE"
"match-found-to-PROB6MULTIPLER"

subject no. 28

nil
nil
nil
nil
nil

nil

subject no. 29

"match-found-to-PROB1TRYONCE"

nil

nil

nil

nil

nil

subject no. 30

nil

nil

nil

"match-found-to-PROB4NORM"

nil

nil

subject no. 31

"match-found-to-PROB1TRYONCE"

"match-found-to-PROB2TRYONCE"

"match-found-to-PROB3TRYONCE"

"match-found-to-PROB4NORM"

"match-found-to-PROB5MULTIPLEF"

"match-found-to-PROB6TRYONCE"

subject no. 32

nil

nil

"match-found-to-PROB3TRYONCE"

nil

"match-found-to-PROB5MULTIPLEF"

nil

subject no. 33

nil

nil

nil

nil

nil

nil

subject no. 34

"match-found-to-PROB1TRYONCE"

"match-found-to-PROB2TRYONCE"

nil

"match-found-to-PROB4NORM"

nil

nil

INTERPRETATION OF OUTPUT

"match-found-to-PROB1TRYONCE"

indicates that on problem one (PROB1) the student's prediction showed the typical error associated with the 'try once and pass' misconception.

"match-found-to-PROB5MULTIPLEF"

would indicate that on problem five the student showed evidence of both the 'try once and pass' error and the 'facts before rules' error.

"match-found-to-PROB4NORM"

indicates that the student on problem four the student gave a correct prediction.

Appendix C3

HAND-ANALYSIS RESULTS 1988 DATA

KEY TO NOTES ON SUBJECTS' ANSWERS:

C CORRECT PREDICTION

F FACTS BEFORE RULES ERROR

M META PREDICTION

A SHORTENED PREDICTION

T TRY ONCE AND PASS ERROR

R RULES BEFORE FACTS ERROR

L INFLUENCED BY LAYOUT

X UNINTERPRETED

Subject 1		Subject 6		Subject 11		Subject 16		Subject 21	
Problem	result	Problem	result	Problem	result	Problem	result	Problem	result
1	T	1	T	1	T	1	TS	1	TM
2	T	2	T	2	T	2	TS	2	TM
3	T	3	T	3	T	3	TS	3	TA
4	C	4	C	4	C	4	C	4	C
5	X	5	TF	5	X	5	TFS	5	X
6	X	6	T	6	X	6	TS	6	X
Subject 2		Subject 7		Subject 12		Subject 17		Subject 22	
Problem	result	Problem	result	Problem	result	Problem	result	Problem	result
1	T	1	T	1	T	1	T	1	T
2	T	2	T	2	T	2	T	2	T
3	T	3	T	3	T	3	T	3	T
4	C	4	C	4	C	4	C	4	C
5	X	5	TA	5	T	5	X	5	TF
6	X	6	T	6	T	6	T	6	T
Subject 3		Subject 8		Subject 13		Subject 18		Subject 23	
Problem	result	Problem	result	Problem	result	Problem	result	Problem	result
1	T	1	T	1	TL	1	T	1	T
2	T	2	T	2	AL	2	T	2	T
3	T	3	TA	3	TL	3	T	3	T
4	C	4	C	4	CL	4	C	4	C
5	X	5	TF	5	TFL	5	T	5	T
6	X	6	T	6	TL	6	T	6	T
Subject 4		Subject 9		Subject 14		Subject 19		Subject 24	
Problem	result	Problem	result	Problem	result	Problem	result	Problem	result
1	X	1	TM	1	T	1	X	1	TL
2	X	2	TM	2	T	2	X	2	TL
3	X	3	TM	3	T	3	X	3	TL
4	C	4	CM	4	C	4	CL	4	CL
5	X	5	X	5	T	5	X	5	TFL
6	X	6	TM	6	T	6	X	6	X
Subject 5		Subject 10		Subject 15		Subject 20		Subject 25	
Problem	result	Problem	result	Problem	result	Problem	result	Problem	result
1	TA	1	T	1	TL	1	T	1	T
2	T	2	T	2	TL	2	T	2	T
3	T	3	T	3	TL	3	T	3	T
4	C	4	C	4	CL	4	C	4	C
5	TF	5	TF	5	TL	5	TF	5	TF
6	T	6	T	6	TL	6	T	6	T

Results of hand-analysis, 1988 data**KEY TO NOTES ON SUBJECTS' ANSWERS:****C** CORRECT PREDICTION**F** FACTS BEFORE RULES ERROR**M** META PREDICTION**A** SHORTENED PREDICTION**T** TRY ONCE AND PASS ERROR**R** RULES BEFORE FACTS ERROR**L** INFLUENCED BY LAYOUT**X** UNINTERPRETED

Subject 26		Subject 28		Subject 30		Subject 32		Subject 34	
Problem	result	Problem	result	Problem	result	Problem	result	Problem	result
1	T L	1	T A	1	X	1	X	1	T
2	T L	2	T A	2	X	2	X	2	T
3	T L	3	T A	3	X	3	T	3	X
4	C L	4	C A	4	C	4	C A	4	C
5	X	5	X	5	X	5	T F	5	X
6	X	6	T A	6	X	6	T A	6	T A
Subject 27		Subject 29		Subject 31		Subject 33			
Problem	result	Problem	result	Problem	result	Problem	result		
1	T	1	T	1	T	1	X		
2	T	2	X	2	T	2	T A		
3	T	3	T A	3	T	3	X		
4	C	4	C A	4	C	4	C A		
5	T	5	T F A	5	T F	5	X		
6	T R	6	T A	6	T	6	X		

Appendix C4

**Self assessed
confidence ratings given
by each student, for
problems one to six in
order shown.**

S 1

almost certainly accurate
almost certainly accurate
probably right
probably right
not certain
not certain

S 2

almost certainly accurate
almost certainly accurate
almost certainly accurate
probably right
probably right
not certain

S 3

probably right
probably right
probably right
probably right
not certain
not certain

S 4

100% accurate
100% accurate
100% accurate
100% accurate
almost certainly accurate
almost certainly accurate

S 5

probably right
100% accurate
100% accurate
almost certainly accurate
100% accurate
100% accurate

S 6

100% accurate
100% accurate
almost certainly accurate
100% accurate
100% accurate
100% accurate

S 7

almost certainly accurate
almost certainly accurate
almost certainly accurate
almost certainly accurate
not certain
not certain

S 8

almost certainly accurate
almost certainly accurate
almost certainly accurate
almost certainly accurate
probably not accurate
almost certainly accurate

S 9

almost certainly accurate
probably right
probably right
almost certainly accurate
probably right
probably right

almost certainly accurate

S 10

100% accurate
probably not accurate
100% accurate
100% accurate
almost certainly accurate
almost certainly accurate

S 15

almost certainly accurate
100% accurate
100% accurate
100% accurate
almost certainly accurate
almost certainly accurate

S 11

100% accurate
100% accurate
100% accurate
100% accurate
not certain
not certain

S 16

100% accurate
100% accurate
100% accurate
100% accurate
almost certainly accurate
almost certainly accurate

S 12

100% accurate
100% accurate
100% accurate
100% accurate
100% accurate
100% accurate

S 17

almost certainly accurate
almost certainly accurate
100% accurate
probably right
probably right
almost certainly accurate

S 13

probably right
probably right
almost certainly accurate
almost certainly accurate
almost certainly accurate
almost certainly accurate

S 18

almost certainly accurate
almost certainly accurate
almost certainly accurate
almost certainly accurate
almost certainly accurate
almost certainly accurate

S 14

almost certainly accurate
almost certainly accurate
almost certainly accurate
almost certainly accurate
almost certainly accurate

S 19

not certain
not certain
not certain
not certain
not certain
probably not accurate

S 20

100% accurate
100% accurate
100% accurate
100% accurate
100% accurate
100% accurate

S 21

probably right
not certain
probably right
probably right
not certain
not certain

S 22

100% accurate
100% accurate
100% accurate
100% accurate
100% accurate
100% accurate

S 23

100% accurate
100% accurate
100% accurate
100% accurate
almost certainly accurate
probably right

S 24

100% accurate
100% accurate
100% accurate
100% accurate

100% accurate

100% accurate

S 25

100% accurate
100% accurate
100% accurate
100% accurate
100% accurate
100% accurate

S 26

probably right
almost certainly accurate
100% accurate
almost certainly accurate
almost certainly accurate
almost certainly accurate

S 27

almost certainly accurate
almost certainly accurate
almost certainly accurate
100% accurate
100% accurate
almost certainly accurate

S 28

almost certainly accurate
almost certainly accurate
almost certainly accurate
almost certainly accurate
almost certainly accurate
almost certainly accurate

S 29

100% accurate
not certain
100% accurate
100% accurate
almost certainly accurate
100% accurate

probably right

S 3 0

probably right
probably right
probably right
100% accurate
probably right
probably right

S 3 1

probably right
probably right
not certain
probably right
probably not accurate
probably not accurate

S 3 2

probably not accurate
probably right
probably right
100% accurate
probably right
probably right

S 3 3

100% accurate
100% accurate
100% accurate
100% accurate
100% accurate
100% accurate

S 3 4

almost certainly accurate
almost certainly accurate
100% accurate
100% accurate
probably right

Appendix C5

Results of machine-analysis of 1987 summer school data

student no. 1

"match-found-to-PROB1NORM"
 "match-found-to-PROB2ONEP"
 "match-found-to-PROB3ONEP"
 "match-found-to-PROB4NORM"
 NIL
 "match-found-to-PROB6ONEP"

student no. 2

NIL
 NIL
 NIL
 "match-found-to-PROB4NORM"
 NIL
 NIL

student no. 3

"match-found-to-PROB1NORM"
 "match-found-to-PROB2NORM"
 "match-found-to-PROB3NORM"
 "match-found-to-PROB4NORM"
 NIL
 "match-found-to-PROB6NORM"

student no. 4

"match-found-to-PROB1NORM"
 "match-found-to-PROB2ONEP"
 "match-found-to-PROB3ONEP"
 "match-found-to-PROB4NORM"
 "match-found-to-PROB5ONEP"
 "match-found-to-PROB6ONEP"

student no. 5

"match-found-to-PROB1NORM"
 NIL
 NIL
 "match-found-to-PROB4NORM"
 NIL
 NIL

student no. 6

NIL
 NIL
 NIL
 NIL
 NIL
 NIL

student no. 7

NIL
 NIL
 NIL
 NIL
 NIL
 NIL

student no. 8

NIL
 NIL
 NIL
 NIL
 NIL
 NIL

student no. 9

"match-found-to-PROB1NORM"
 "match-found-to-PROB2NORM"
 "match-found-to-PROB3NORM"
 "match-found-to-PROB4NORM"
 "match-found-to-PROB5NORM"
 "match-found-to-PROB6NORM"

student no. 10

NIL

NIL

"match-found-to-PROB3TRYONCE"

"match-found-to-PROB4NORM"

NIL

NIL

student no. 11

NIL

NIL

NIL

"match-found-to-PROB4NORM"

NIL

NIL

student no. 12

"match-found-to-PROB1NORM"

"match-found-to-PROB2NORM"

"match-found-to-PROB3ONEP"

"match-found-to-PROB4NORM"

"match-found-to-PROB5ONEP"

"match-found-to-PROB6ONEP"

student no. 13

NIL

"match-found-to-PROB2NORM"

NIL

NIL

NIL

NIL

student no. 14

NIL

NIL

NIL

NIL

NIL

NIL

student no. 15

"match-found-to-PROB1NORM"

"match-found-to-PROB2NORM"

"match-found-to-PROB3ONEP"

"match-found-to-PROB4NORM"

"match-found-to-PROB5ONEP"

"match-found-to-PROB6ONEP"

student no. 16

NIL

NIL

NIL

NIL

NIL

NIL

student no. 17

"match-found-to-PROB1NORM"

NIL

"match-found-to-PROB3NORM"

"match-found-to-PROB4NORM"

NIL

"match-found-to-PROB6NORM"

student no. 18

NIL

NIL

NIL

"match-found-to-PROB4NORM"

NIL

NIL

student no. 19

NIL

"match-found-to-PROB2ONEP"

NIL

"match-found-to-PROB4NORM"

NIL

"match-found-to-PROB6ONEP"

student no. 20

NIL
 NIL
 NIL
 NIL
 NIL
 NIL

student no. 21

"match-found-to-PROB1TRYONCE"
 "match-found-to-PROB2TRYONCE"
 "match-found-to-PROB3TRYONCE"
 "match-found-to-PROB4NORM"
 "match-found-to-PROB5MULTIPLEF"
 "match-found-to-PROB6TRYONCE"

student no. 22

NIL
 NIL
 NIL
 "match-found-to-PROB4NORM"
 NIL
 NIL

student no. 23

NIL
 NIL
 NIL
 "match-found-to-PROB4NORM"
 NIL
 NIL

student no. 24

NIL
 NIL
 NIL
 "match-found-to-PROB4NORM"
 NIL
 NIL

student no. 25

"match-found-to-PROB1NORM"
 "match-found-to-PROB2ONEP"

"match-found-to-PROB3ONEP"
 "match-found-to-PROB4NORM"
 "match-found-to-PROB5ONEP"
 "match-found-to-PROB6ONEP"

student no. 26

NIL
 NIL
 NIL
 "match-found-to-PROB4NORM"
 NIL
 NIL

student no. 27

NIL
 NIL
 NIL
 "match-found-to-PROB4NORM"
 NIL
 NIL

student no. 28

NIL
 NIL
 NIL
 "match-found-to-PROB4NORM"
 NIL
 "match-found-to-PROB6ONEP"

student no. 29

NIL
 NIL
 NIL
 "match-found-to-PROB4NORM"
 NIL
 NIL

student no. 30

NIL

NIL

"match-found-to-PROB3ONEP"

"match-found-to-PROB4NORM"

NIL

"match-found-to-PROB6ONEP"

student no. 31

"match-found-to-PROB1NORM"

"match-found-to-PROB2NORM"

"match-found-to-PROB3NORM"

"match-found-to-PROB4NORM"

"match-found-to-PROB5FACTS"

"match-found-to-PROB6NORM"

student no. 32

NIL

NIL

NIL

NIL

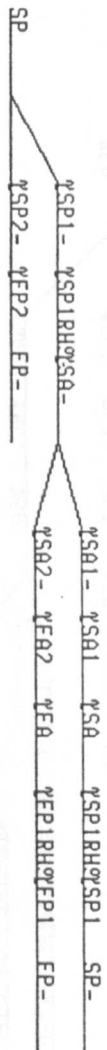
NIL

NIL

/users/pat/psdlr/pifa_st.tmp

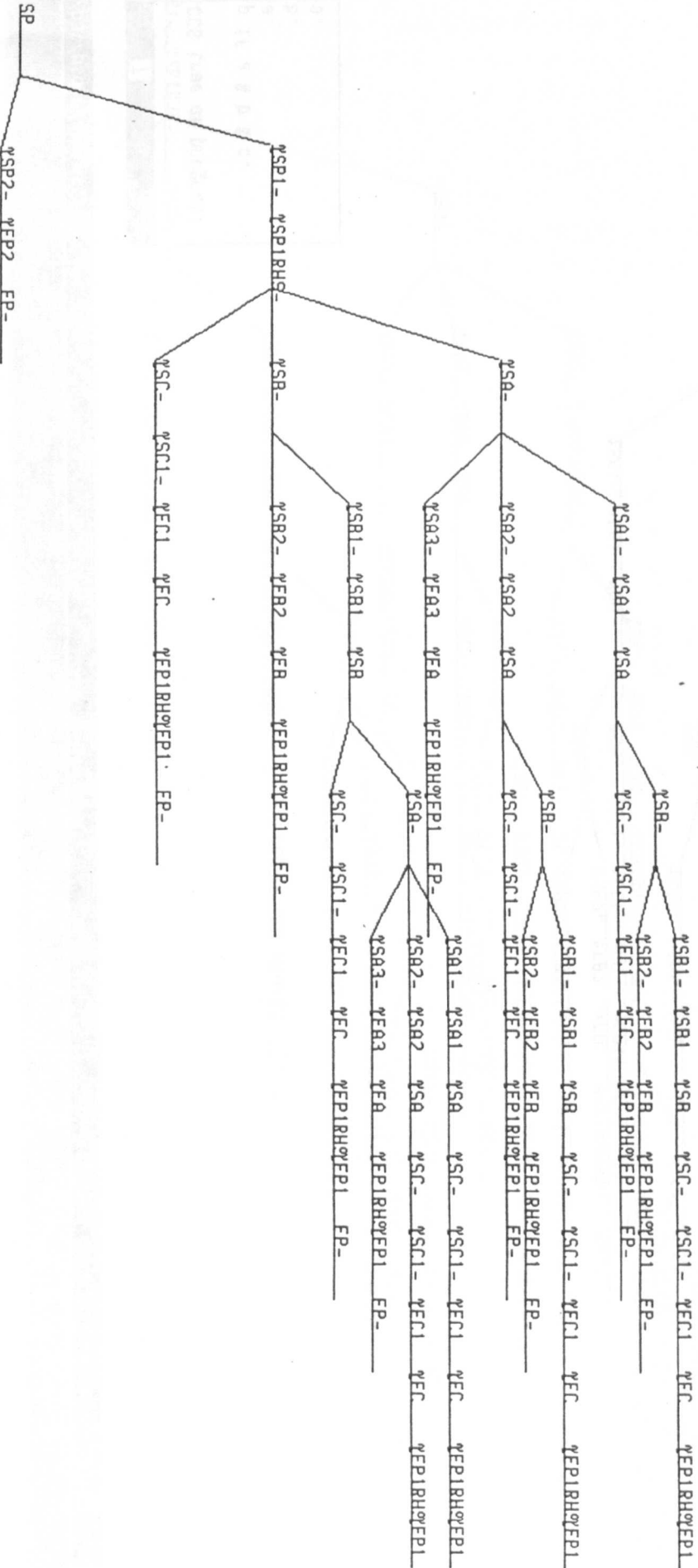
I

I

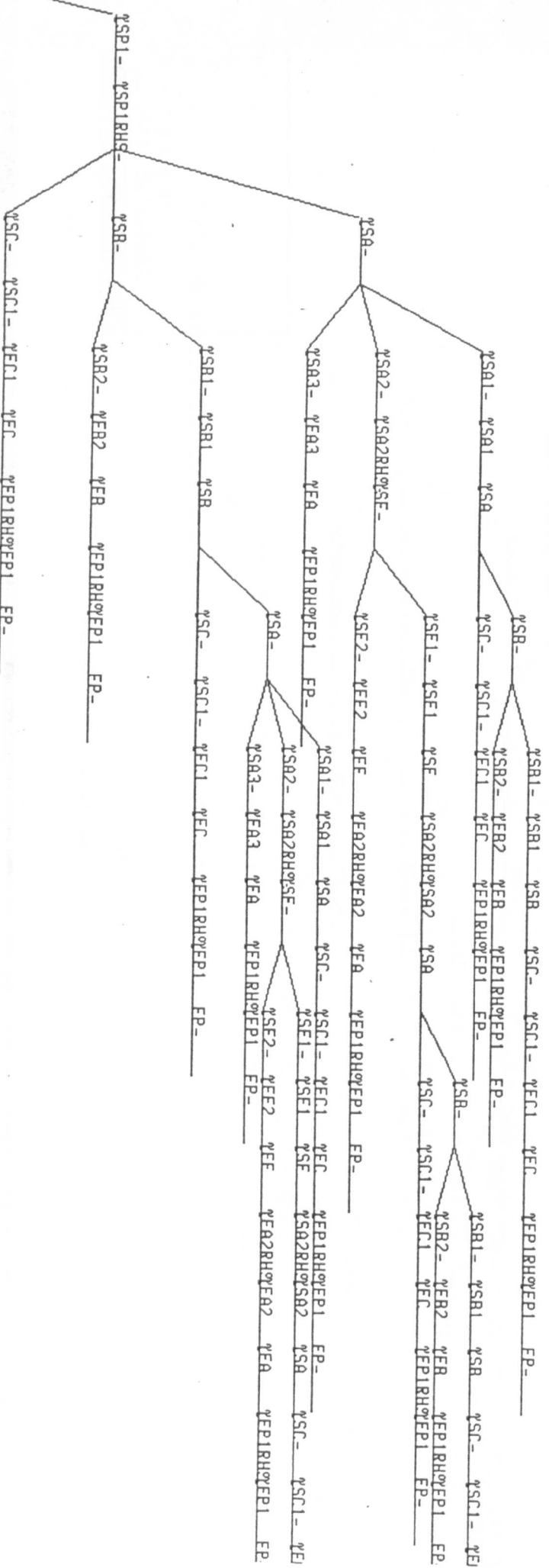


I
 CCS tree for program:
 p if a.
 a.

I
 1
 CCS tree for program:
 p if a & b & c.
 a.
 a.
 b.



1
p if a & b & c.
a.
b.
a if e.
e.



CENTRE FOR INFORMATION TECHNOLOGY IN EDUCATION

List of CITE Ph.D. Theses

<u>Thesis No.</u>	<u>Title and Author</u>
1	Jenny Preece, (March 1985) Interpreting Trends in Graphs: A Study of 14 and 15 Year Olds. (Available from Open University library)
2	Tony Priest, (October 1986) Modelling Student Errors in Physics Problem-Solving. (Available as CAL Technical Report No. 68)
3	Alistair Edwards, (July 1987) Adapting user interfaces for visually disabled users. (In press by Paul Chapman's Publishers)
4	Simon Holland, (July 1989) Artificial Intelligence, Education and Music. The use of Artificial Intelligence to encourage and facilitate music composition by novices. (Available as CITE Report No. 88)
5	Pat Fung, (October 1989) An Application of Formal Semantics to Student Modelling: an investigation in the domain of teaching Prolog.