An in-depth study of the cognitive behaviour of novice programmers

Thesis

AN IN-DEPTH STUDY OF THE COGNITIVE BEHAVIOUR OF NOVICE PROGRAMMERS

JOSEPH HENRY KAHNEY, B.A., Ph.D.

Thesis submitted in partial fulfillment of requirements for Ph.D. in Psychology, 15th December, 1982.

Date of submission: 1·12·82

Date of award: 22·12·82

DEDICATION

This thesis is dedicated to my sons,
Leander, Alexander, and Christopher,
because they are beautiful,
and to John Pickering, for his kindness.

ACKNOWLEDGEMENTS

I owe my greatest debt of gratitude to my thesis supervisor, Marc Eisenstadt. He gave me both intellectual and financial support, and supplied his time and editing skills in a measure that would make any other research student sinfully envious.

I want to thank Henry Rueter for discussions on the Sorting and the Recall tasks described in Chapter 3 of this thesis.

I want to thank Peter Whalley, Martin LeVoi, and Martin Conway for their perseverence in discussing statistical methods with me.

I want to thank Martin LeVoi also for keeping the PDP-11 switched on well beyond its time, just for the sake of this thesis.

I want also to thank Tim O'Shea whose comments on my research have always been very helpful to me.

I want to thank Tony Hasemer for creating a program which allowed me to translate thousands of meaningless numbers into readable data.

I want to thank Rick Evertsz for his skill and patience in implementing the production systems presented in the thesis.

I want to thank Jon Slack, who made it all possible for me in the first place.

I thank Leander Kahney for the beautiful trees and graphs he drew for the thesis, and the hundred and one other jobs he did for me.

I thank Mary Bailey for double-checking my categorization of novice programs, and for creating some of the Figures and Tables that appear in the thesis.

I thank Jane Price for transcribing tapes, drawing graphs, and any number of typing tasks.

I thank Janice Simpson for transcribing tapes and editing some of my computer files for me.

I thank Fiona Spensley for an unbelievable amount of painstaking and good natured help in the last weeks of preparation of the manuscript.

I thank Danny Waite for his fault-finding skills. Few departments can have acquired a technician of his calibre.

I thank Pat Vasiliou and Ortenz Rose for their secretarial help over the past three years.

I thank Doreen Warwick for her help with the transcription of tapes.

ABSTRACT

This thesis reports a series of studies of the behaviour of novice computer programmers. One hundred and thirty nine programs which had been designed as solutions to a particular programming problem were analyzed, and it is shown that despite great surface variety, the programs exhibit considerable underlying order. Concept Sorting and Recall tasks were used to induce the novices' mental organizations of programming knowledge. A program Questionnaire was designed which presented Subjects with alternative possible programmed solutions to a particular problem. The novices' mental models of the behaviour of recursive procedures were determined from their selections, and from the protocols they provided on the predicted behaviour of the different programs. In order to determine the structure perceived by novices when they were presented with unfamiliar programs, a transcription task was designed. The role of A) real world knowledge, and B) programming knowledge in problem understanding processes was investigated, and an 'interactionist' theory of problem solving in computer programming put forward. The pattern of problem solving processes predicted by the theory was formalized in an Interpretation Theory for scoring verbal protocols. The theory was tested by presenting novices with a problem statement containing some statements which were designed to activate real world knowledge, and others that were designed to activate programming knowledge. The anlaysis of protocols given by two novices is provided, and, finally, production system models of some aspects of their program writing behaviour are provided.

TABLE OF CONTENTS

FIGURES.

CHAPTER 2:

CHAPTER 3:

CHAPTER 4:

CHAPTER 5:

CHAPTER 6:

CHAPTER 7:

CHAPTER 8:

CHAPTER 1

THE BEHAVIOUR OF NOVICE AND EXPERT PROBLEM SOLVERS

1.1 INTRODUCTION

The goal of this thesis is to provide a model of novice programming behaviour. Although some work has begun on novice programmers, the models that have resulted have little to say about what the novice programmer knows, or how he uses what he knows. By contrast, models of novices in the domain of physics are fairly powerful, and are presently being further elaborated (Larkin, McDermott, Simon & Simon, 1980; Chi, Feltovich & Glaser, 1982). Why is this? It is not because more people are working on modelling novice behaviour in the latter domain. Rather it is because those who model novices learning physics have adopted an already elaborated model of physics experts. The model of the physics expert (Bhaskar & Simon, 1977;) in turn, was borrowed from the research that had been done on human problem solving, particularly on experts in chess (Newell & Simon, 1972; Chase & Simon, 1973). Basically, the 'chess expert' model suggests that the expert has massive amounts of highly structured knowledge about chess positions, and that a subset of this knowledge is triggered by patterns existing on a chess board at a particular point in time. Once triggered, this relevant subset of

Knowledge makes available a response to the situation that exists, or a candidate set of responses which can be used in a further search of the problem space so that the best of the candidate responses may be determined (Chase & Simon, 1973; Simon & Gilmartin, 1973). The model provides a powerful framework for thinking about many different aspects of problem solving tasks. For example, under the model there are quite complex notions of 'the organization of Knowledge'. Simon & Simon include under this label the access, or indexing, characteristics to a fragment of Knowledge, arguing that as well as having more Knowledge, the expert has a vast number of access routes to a particular fragment of Knowledge. That is, both expert and novice may share Knowledge relevant to a task in hand, and yet the novice may fail to use it because it was not triggered by aspects of the current task, or because other powerful variables preclude use of such Knowledge even if it is accessed (see, for example, the comments on the Adelson study, below). In the study of what it is that experts and novices Know and do in the domain of physics, this 'chess expert' model has been adopted wholesale, like a predicate that would be given new arguments (Knowledge of physics in the place of Knowledge of chess). Overall, the adoption has been a success. Problem understanding processes have been been elaborated (Novak, 1977; McDermott & Larkin, 1980; Chi, Feltovich, & Glaser, 1982; Heller & Reif, 1982); the Knowledge base on which problem solving processes draw have been delineated for various problem types (Larkin, et al., 1980; Chi, Feltovich & Glaser, 1982); and process models of strategies reflecting the behaviour of both novices and experts have been implemented in computer programs whose traces closely reflect (with a few adjustments) the strategies used by solvers with different abilities (Larkin, et al., 1980). One thing that this thesis will show is that this same basic framework serves the study of novice programmers. It will also show that in order to devise an accurate portrayal of the novice programmer, different methodologies than those currently being used are necessary, and also that it is important to drop the illusion of the average novice. Brooks (1977) has shown quite convincingly that the idea of the prototypical expert is an illusion, and so it is obviously a mistake to suggest that these widely variant products of long and varied training experiences arise from a common stock. Methodologies currently being used permit only glimpses at isolated aspects of the phenomena of interest.

1.2 WHAT DO WE KNOW ABOUT NOVICE PROGRAMMERS?

Much of what we already know about either novice or expert computer programmers can be summarized in the brief statement: experts know more than novices, and what they know is better organized. This is clearly a low level of explanation. The hypothesis is precisely stated by Mayer (1981):

> What do experts know about computer programming that beginners do not know? One answer is that experts possess much more information and that the information is organized more efficiently.

To varying degrees, models of novice programmers are 'know nothing', or 'empty head' models. They stress what knowledge the novice has not got. According to one investigator - (Adelson, 1981) - novices have not got 'functional' organizations of programming knowledge. According to others - (McKeithen et al., 1981) - novices have not yet acquired the conceptual chunks that experts are shown to possess. A third model - (Soloway et al., 1982; Ehrlich & Soloway, 1982) - is based on the novices lack of 'tacit plan Knowledge'. To be fair, all the above named researchers do attribute some Knowledge as being inside the heads of novice programmers, but the stress in the first two studies is put on what experts can do and Know, and, at least by implication, what novice's cannot do and have yet to learn. Adelson portrays novices as being fixated at the syntactic stage of programming Knowledge. This stage apparently lasts over a considerable period of time: the 'novices' studied by Adelson differ greatly in the amount of experience they have with programming, but experience doesn't seem to do them much good. The process of becoming an expert, according to Adelson's results, is more like a metamorphosis than a gradual process.

McKeithen et al. portray novices as freeing themselves, or their newly acquired programming concepts, from common language associations and gradually acquiring the domain specific meanings of those concepts. The transition from beginner's status to that of expert, indicated by McKeithen's results, is very different from that suggested by Adelson.

The single serious attempt to attribute detailed Knowledge to novices - even though the Knowledge is assigned to them by default (see below) - is that of Soloway and his group at Yale. They show that a meaningful task like extracting information from an incomplete program, and using the extracted information in combination with their own Knowledge to fill in the missing lines of the program, experts do very well by comparison with novices. But, of course, as the Soloway group point out, the experts have had a lot more practice at this than have the novices. The focus in their studies is on what practitioners at any skill level are capable of achieving and what exactly they do know. The approach involves working out in some detail the kind of knowledge a programmer must have in order to achieve a certain programming effect, and then testing to see how many programmers, of whatever level of skill, can achieve that effect. In short, if a person can solve a problem requiring a particular piece of knowledge, then that person (by default) must possess that knowledge.

There is an interesting relationship between the power of the various models of the novice programmer and the kind of experimental work done to investigate them. A fairly meaningless task produces a weak, almost certainly inaccurate, or misleading model of what the novice is like (Adelson). A somewhat more sensible task at least confirms our simplest intuitions about the growth of knowledge (McKeithen). And a task that is relevant to the usual occupations of both novice and expert programmers is most revealing of all (Soloway).

Adelson (1981) used a multitrial free recall task (MFR) in order to induce the underlying organizations of knowledge of her subjects from their performance on the task. Adelson presented subjects with 16 scrambled lines of PPL code from three structured sorting routines. Subjects, both novices and experts, viewed each of the 16 lines of code for 20 seconds and when all 16 lines had been seen they were given eight minutes to recall all they could. Then a second trial was begun. In all, subjects went through eight such trials. Adelson was able to show that in the end experts recalled together the lines that belonged together in one or another of the original three sorting routines. Novices recalled lines by syntactic class. Adelson argues that the experts use "program membership as a basis for the inclusion and exclusion of items". Essentially, what experts have is an hierarchically organized set of relational information structures, or schemas, and it is these functional structures that give experts their advantage over novices on a wide range of tasks. In a recall task such as this, experts are presumed to use their knowledge of (various) sorting routines for assimilation of the various lines of code, allowing them not only to determine which lines 'go together' but also to determine the correct order of the lines within each program. Novices, on the other hand, have acquired some knowledge of the syntax of the language, and organize the presented material in terms of syntactic class membership. The explanation, in other words, is that experts know more, and what they know is better organized. The explanation is one variant on the summary statement quoted from Mayer, above, and constitutes what would simply be a global, or descriptive - non explanatory - statement under a problem solving model.

Adelson's results point to two distinct classes of programmers: the 'haves' (experts) and the 'have nots' (novices). Adelson provides a long catalogue of differences between novices and experts, including: amount that was recalled as a result of the experimental manipulations, degree of subjective organization, and manner of organization. Adelson's novices included trainees with different amounts of experience. One group, the 'short-term' novices, were 'currently enrolled in their first course in programming'; the 'long-term' novices had completed 'more than two programming courses'. Even so, the long-term novices behaved exactly as the short-term novices on the recall task - experience doesn't count for much in PPL programming.

Further, as already indicated above, the results suggest that the transition from novice to expert is more a metamorphosis than a gradual acquisition of skill: Adelson was unable to identify any point at which novices might be gaining competence in programming.

The finding that long-term novices still behave like absolute beginners may have something important to say about the teaching of these novices. A more likely explanation is that the task disguises what novices know, rather than reveals their knowledge of programming. It is inconceivable that these novices are able to write only one line programs, as suggested by these results (as indeed Adelson herself suggests in her comparison of novices and experts, p. 431).

An examination of the code presented in Adelson's experiment shows that the parameter and local variable names for each of the three programs are different, and that each line of each different program contains either the parameter name or one of the local variable names, and these are features presumably used extensively by the experts in determining which lines of code belonged to which programs. For novices, understanding a line of code, and its relationships to others involves reasoning rather than recognition. Their set task was not to reason about the lines presented, but to commit them to memory for later recall. The fact that they use syntactic class as an organizing principle in commiting the 16 lines of code to memory should not preclude their ability to see other possible organizing principles. In order to discover what abilities the novices actually do possess, Adelson might profitably have asked her novices, once they had completed the MFR task, if they knew any other ways in which the stimulus set could have been organized. Soloway has shown that novices - even 'short-term novices' - have a sophisticated ability to work with variables which suggests that Adelson's novices, even if they had not reconstructed the programs as originally written, should be able - using variable names as indices - to indicate which lines belonged to different programs. In Soloway's study (see below) novices had to infer the purpose of the code that was presented in order to fill in the blank lines, indicating that novices are capable of understanding previously unseen programs, and so, presumably, Adelson's novices also could indicate in some general way the behaviour of the programs that resulted from their re-organization (had reorganization been permitted) of their recall orders. Admittedly, the ability of novices to recognize possible orderings other than those actually produced is hypothetical. But if the ability had been shown to exist - which is likely, given Soloway's results - then the fact that experts are able to utilize higher order knowledge in conditions where novices cannot, even though they have it, raises some important issues about the 'differences' between members of groups at different skill levels. For instance, if both novices and experts 'have' the same knowledge, what exactly are the variables, in a current situation, that prevent one group from using this knowledge? What are the causes of the effects of these variables?

Another study involving novice, intermediate and expert programmers is that by McKeithen, Reitman, Reuter & Hirtle (1981). McKeithen et al. use the familiar technique of first demonstrating reliable differences between groups of subjects on one kind of task - and then attempting to demonstrate or indicate the underlying 'causes' of the observed differences from the results of a second task. McKeithen et al. begin with a demonstration that there are statistically significant differences between the performance of expert programmers, intermediates, and beginners in recalling an unfamiliar program, after brief periods of viewing the program. Their explanation for the differences is - again - in terms of the greater amount of information the expert has and the organization of this greater amount of knowledge. Experts are conceived to have 'chunked' their knowledge, and elementary chunks are integrated together into higher order chunks (at this level, the explanation is no different from Adelson's, described above). The more chunks a person has, and the better structured those chunks, the less problem solving required of the person performing on a particular task in the domain in which he is knowledgable. It is the same explanation used by Chase & Simon (1973) to explain the performance of chess masters in identifying meaningful configurations of pieces on a chessboard. McKeithen et al. tested the hypothesis using a multitrial free recall task. Subjects of different ability levels were asked to memorize a list of Algol-W key-words and, when subjects were able to recall the list of concepts twice in succession without error, the data collection phase of the experiment began. Subjects then recalled all the concepts over a number of trials, each of which began from a different element of the learning set; every element of the set served as a starting point over 21 trials. There were also a few trials interspersed in which subjects recalled from any starting point within the set that they themselves chose. From the entire set of recall orders the subjects' subjective organizations of the concepts were derived and represented as directed graph structures using the Reitman & Reuter directed tree analysis technique (Reitman & Reuter, 1980).

McKeithen's results indicate that experts know more or less the same things, and know them in the same way (many chunks are shared) whereas the novices know less and what they know is idiosyncratic (few chunks are shared). McKeithen's intermediates shared programming knowledge with experts more than with one another. That is, experts tend to recall together concepts that function together in a program - WHILE-DO, for example - whereas everyday associations determine which concepts intermediates and to a lesser extent novices and experts will recall together. McKeithen's results, however, also reveal a number of similarities amongst the three groups studied. For example, the amount and depth of organization was the same for all groups, i.e., amount and depth of organization of domain specific conceptual knowledge is invariant over a lengthy learning period such as that governing the transition from novice to expert status.

It is of course not surprising that experts in the McKeithen study know more than novices most of the concepts included in the recall list would not have been introduced to novices in terms of their meanings in Algol-W programming (the novices were in the first week of their first programming course) so everyday associations are all that novices have as a means of organizing the material. The value of the experiment is the indication it gives of the way this initial organization changes over the duration of a single course in programming. At the end of their first programming course, the novices in the McKeithen study have begun to acquire some of the chunks of the experts (which, like Soloway's findings, and unlike Adelson's, indicates a gradual acquisition of knowledge rather than a sudden restructuring only after a very lengthy learning period).

An entirely different approach to the study of novice/expert knowledge structures has been taken by Soloway's group. They argue (Ehrlich & Soloway, 1982) that experts have tacit plan knowledge - script like representations of the stereotypic actions in programs - which is developed from long programming experience. Since the knowledge is tacit, experienced programmers wouldn't be expected to be able to specify exactly what knowledge they were using in solving problems. This being the case, Soloway's group have created schema like representations for some of the knowledge - e.g., variable plan knowledge - based on their intuitions about what information must be contained in tacit plans. They then use the plans as a basis for devising programs having one or more blank lines, which programmers of different levels of experience are asked to study and fill in. Experts are significantly better at filling the blanks with appropriate program statements. One explanation for the results is that the experts have the specific knowledge contained in the tacit plans devised by the investigators. Presumably, by extension, those novices who solve the particular problems solved by experts, share a fragment of tacit knowledge with experts - the fragment (or plan) necessary to solve the particular problem. To be fair to them, Ehrlich & Soloway do not specifically claim that novices and experts share a particular tacit plan if they can solve the same problem. Indeed, in the introduction to their paper they claim to be in the business of identifying "specific knowledge which experts seem to have and use, and which novices have not yet acquired". Yet, in the body of the paper they say of one experimental result: "...our data suggest that non-novice programmers have no difficulty in recognizing that the Counter Variables need to be initialized via an assignment statement..... Thus they performed as if they did understand the relationship between initializing and updating Counter Variables. The data also suggest that most, but by no means all, of the novices also understand this relationship." Whatever the precise nature of the argument might turn out to be, the results of one of the studies presented in this thesis indicates that there are programming problems which both novices and experts could solve, but that the solutions would be based in entirely different knowledge bases (see Chapter 4). The lesson to be learned from the experiment to be discussed later in this thesis is that a group of solvers should not be attributed knowledge structures by default, as would seem to be the case in the Ehrlich & Soloway study. The major problem with the study is that the focus of attention is still on the expert, the object is to specify plan knowledge related to expertise. In order to discover what knowledge novices actually have, experiments should be designed which have novices at the centre of attention.

Finally, other research into the behaviour of computer programmers either deals only with experts (Brooks, 1977; Schneiderman, 1982) or with differences between novices and experts on isolated processes (Jeffries, 1982). Their models will be discussed at relevant points in the Chapters that follow.

SUMMARY: The framework in which recent investigations of novice computer programmers has been carried out has provided us with some valuable information about some of the knowledge expert computer programmers have, and how their knowledge is organized. However, the results are, at best, only suggestive of the knowledge novices have; arguments have been put forth showing that there are problems with accepting the findings at face value. All the studies presented in this thesis have as their goal the specification of the knowledge which novices posses. There are also attempts to compare and contrast this knowledge with the knowledge of experts, but these attempts are always made with an eye towards specifying what it is that novices know.

1.3 AN INTRODUCTION TO SOLO PROGRAMMING

Familiarity with some of the SOLO programming language is prerequisite for an easy understanding of many of the experiments and the discussions contained in this thesis. I shall introduce the SOLO primitives in terms of program 'segments' - the main groupings of program statements into which students' programs can be analysed. The notion of program segments which have their own stereotyped behaviour is derived from the work of Rich & Shrobe (1978) and Waters (1982). Indeed, some of the terminology below is based on the 'plan libraries' which Rich, Shrobe, and Waters have posited to be part of the repertoire of experienced LISP programmers. Although SOLO novices can not necessarily articulate the nature of these program segments, the segments are clearly exhibited in many novices' programs as we shall see below.

There are only a handful of primitives in SOLO. A brief description of their functions and their implementation in SOLO will be given in this section. The language is described in detail in a purpose-written Programming Manual (Eisenstadt, 1978), and the design features which make it particularly suitable for novices are described in (Eisenstadt, 1982).

SOLO is a database manipulation language utilizing a small number of primitives for accessing, adding and deleting unidirectional, associative triples of the form 'node--relation--node', e.g.:

```
JOHN ISA MAN
```

or

```
FIDO HAS FLEAS.
```

1.3.1 Side-Effect segments: using NOTE and FORGET

A triple may be added to the database using the primitive NOTE, e.g.:

```
NOTE JOHN LOVES MARY
NOTE JOHN SMOKES CIGARS
NOTE JOHN OWNS FIDO
```

Assuming that the triples 'JOHN ISA MAN' and 'FIDO ISA DOG' already existed in the database, the additions would give the following database:

```
JOHN ISA MAN
 |
 |...LOVES...MARY
 |
 |...SMOKES...CIGARS
 |
 |...OWNS...FIDO
FIDO ISA DOG
```

Deletion from the database is effected with the FORGET primitive, e.g.:

```
FORGET JOHN LOVES MARY
```

the result of which would be the database:

```
JOHN ISA MAN
 !
 !...SMOKES...CIGARS
 !
 !...OWNS...FIDO

FIDO ISA DOG
```

Both NOTE and FORGET have side-effects on the existing database, and as such are referred to, when they are used in programs, as 'Side-effect segments'. These segments may be either 'conditional' (Conditional side-effect segments) or 'unconditional' (Unconditional side-effect segments), as explained below.

1.3.2 The Conditional segment: using CHECK

SOLO conditionals permit branching in a program and are implemented with the primitive CHECK. The framework of a CHECK segment is this:

```
CHECK <triple>
If Present: <action 1> ; CONTINUE or EXIT
If Absent: <action 2>; CONTINUE or EXIT
```

The programmer is automatically cued with the 'If Present' prompt after he has written 'CHECK <triple>' followed by a carriage return. Each branch can carry one instruction (nesting of CHECKs is not permitted) and must contain a flow of control statement - either 'CONTINUE' or 'EXIT'. If one of these does not appear before a carriage return, the programmer is prompted with a message to produce one.

An example use of the use of CHECK is this. Suppose we wanted to make the inference that if some person P smokes cigars, that person is unhealthy, but otherwise the person is okay.

The primitive 'TO' allows us to define a procedure in SOLO. In order to define a procedure called 'HEALTHASSESS' with one parameter, we would write:

```
TO HEALTHASSESS /X/.
```

The first step of the procedure, using the CHECK statement described above, would be:

```
1 CHECK /X/ SMOKES CIGARS
1A If Present: NOTE /X/ IS UNHEALTHY; EXIT
1B If Absent: NOTE /X/ IS OKAY; EXIT
```

The end of a procedure definition is indicated with the primitive 'DONE'. The entire procedure, then, would be:

```
TO HEALTHASSESS /X/
1 CHECK /X/ SMOKES CIGARS
1A If Present: NOTE /X/ IS UNHEALTHY; EXIT
1B If Absent:  NOTE /X/ IS OKAY; EXIT
DONE
```

The nodes in a CHECK statement may also be variables. Wild-card pattern matching is allowed on the right hand node only, the special wild-card symbol being "?". If a pattern is found in the database that matches, the value of the right hand node is then automatically bound to the symbol * (star variable), which can then be referred to in subsequent processing. For example, suppose we wanted to create a procedure called 'INFORM', which told us, for some 'X', the value of the node linked to 'X' via the 'ISA' relation - if any. We could write:

```
TO INFORM /X/
1 CHECK /X/ ISA ?
1A If Present: PRINT /X/ "HAPPENS TO BE A" *; EXIT
1B If Absent: PRINT "I DON'T KNOW WHAT" /X/ "IS. SORRY"; EXIT
DONE
```

If our database contained only the following triples:

```
FIDO...ISA...DOG
 |
 |.....EATS...MEAT
```

and we ran 'INFORM' with 'FIDO' as the value of the parameter, then * would be bound to DOG. The result of running the procedure would be the printed string:

```
FIDO HAPPENS TO BE A DOG
```

If, on the other hand, we ran 'INFORM' with 'ROVER' as the value of the parameter we would get the printed string:

```
I DON'T KNOW WHAT ROVER IS. SORRY.
```

because the pattern match would fail and the 'If Absent' branch of the conditional statement would be taken.

1.3.3 The Conditional side-effect (CSE) segment: combining NOTE and CHECK

The conditional side-effect segment adds (deletes) triples conditional upon the presence (absence) of another triple (or configuration of triples) in the database. That is, in order to add the inference that some 'X' is guilty if 'X' is a burglar, the SOLO programmer could write a program called 'GUILTASSESS' such as this:

```
TO GUILTASSESS /X/
CHECK /X/ ISA BURGLAR
If Present: NOTE /X/ IS GUILTY; EXIT
If Absent:  EXIT
DONE
```

1.3.4 The Generate Next Object & Self (GNO/S) segment

Generate Next Object and Self is the name given to a segment of program designed to trigger recursion, and contains two parts. The 'Generate Next Object' part of the segment uses wild-card pattern-matching to generate successive nodes from a database for the procedure to operate upon, and the 'Self' part takes the name of the procedure being used recursively. For example, suppose we wanted a procedure, called 'INFECT', which gave 'flu to some arbitrary node, and then spread the infection to all nodes along a chain of 'KISSES' relations. The nodes in this chain will be referred to throughout the thesis as a Chain Relation List. We could write:

```
TO INFECT /X/
NOTE /X/ HAS FLU
CHECK /X/ KISSES ?
If Present: INFECT *; EXIT
If Absent: EXIT
DONE
```

Given the database:

```
JOHN....ISA....MAN
 !
 !......KISSES....MARY

MARY....ISA....WOMAN
 !
 !......KISSES....TIM
```

```
TIM....ISA....MAN
```

and the 'INFECT' procedure, the result of running the program with 'JOHN' as the value of the parameter, would be the following:

```
JOHN.....ISA....MAN
 ¦
 ¦......KISSES....MARY
 ¦
 ¦......HAS....FLU
MARY....ISA....WOMAN
 ¦
 ¦......KISSES....TIM
 ¦
 ¦......HAS....FLU
TIM....ISA....MAN
 ¦
 ¦.....HAS....FLU
```

That is, at the first step of 'INFECT' the side-effect on 'JOHN' would occur. The database contains 'JOHN KISSES MARY', so at step 2 * would be bound to 'MARY' and at step 2A a recursive call to the 'INFECT' procedure, with 'MARY' as the parameter value, would be triggered. The same process would occur for 'TIM', after which the program would be terminated (step 2B).

1.3.5 The FILTER segment

A FILTER is a special type of Conditional segment, and is used to detach one of the nodes from a Chain Relation list. The idea is to treat the filtered node separately from the other nodes on the Chain Relation list. We will be concerned with FILTERs that are used as a condition for triggering a subprocedure. Suppose, for example, we wanted to write an 'INFECT' program which spread illness along a chain of nodes having a 'KISSES' link, but only if the original parameter already has a disease. We could write a FILTER program which tested for disease on the original node, and a subprocedure which spread infection recursively. Call the FILTER procedure 'DISEASETEST' and the subprocedure 'INFECT'. We would write:

```
TO DISEASETEST /X/
CHECK /X/ HAS DISEASE
If Present: INFECT /X/; EXIT
If Absent: EXIT

TO INFECT /X/
1 NOTE /X/ IS ILL
2 CHECK /X/ KISSES ?
2A If Present: INFECT *; EXIT
2B If Absent:  EXIT
DONE
```

The reader is left to simulate the behaviour of the program - given the database following this sentence - to test his understanding of the SOLO language from the introduction provided here.

```
JOHN....ISA....MAN
 !
 !......HAS....DISEASE
 !
 !......KISSES....MARY

MARY....ISA....WOMAN
 !
 !......KISSES....TIM

TIM....ISA....MAN
```

The reader is invited to predict what would happen if we now typed in DISEASETEST JOHN.

1.4 SUMMARY OF THE STUDIES REPORTED IN THIS THESIS

An overview of the experimental tasks and simulation models that are to be discussed in detail within the next seven Chapters of the thesis is presented below. The following information is provided for each task:

1) a brief description of the task, and an overview of the findings;

2) the Subjects;

3) the data that was used for analysis;

4) the analyses that were carried out;

The novices were all students taking the third level course 'Cognitive Psychology'. (The course will hereafter be referred to as 'D303', which is the Open University course number for 'Cognitive Psychology'.) They also all acted as Subjects in other tasks that are outlined below. The exception (S8) was a new post-graduate student at the Open University, embarking on research in text-studies, and had had no previous programming experience. S8 was treated like all other novice Subjects: required to read the course material and to perform all the 'Study Centre' activities which D303 students are required to perform. The Subjects are called S5, S6, S7, S8, S9, and S10. The experts also acted as Subjects in many of the tasks outlined below, although it was not always possible to use the same experts in all of the tasks. The experts are called S13, S14, S15, S16, S17, S18, S19, and S20. All the experts had had experience in tutoring in the artificial intelligence projects at the D303 summer schools, which required them to help students design and implement SOLO programs. All also had programming experience in at least one other programming language.

1.4.1 Order out of chaos: analysis of 139 programs

One hundred and thirty nine programs which had been designed as solutions to a particular programming problem were analyzed, and it is shown that despite great surface variety, the programs exhibit considerable underlying order. Sixty five percent of the programs can be assigned to only five different classes of programs. These programs are related to the problem statement in interesting ways. The interpretation of the data suggest that novices experience a lot of difficulty making their programs reflect the problem models they have in their heads. Moulding data structures to program designs is one solution to this problem. The particular problem is described in detail in Chapter 2.

SUBJECTS: 139 Open University undergraduates studying a third level course in Cognitive Psychology (Course Number D303). Nothing is known about the computing (or any other) background of any of the project students.

DATA: Final program.

ANALYSES: Classification of data structures. Classification of program structures.

1.4.2 Internal organization: sorting and recall of programming concepts

Concept Sorting and Recall tasks were used to induce the novices' mental organizations of programming knowledge. The important finding was that there are no significant differences between novices and an 'ideal' expert with respect to the underlying organizations of conceptual (domain related) knowledge. These are reported in Chapter 3.

SUBJECTS: Six novices and seven experts acted as Subjects in the Sorting task (Novices = S5, S6, S7, S8, S9 and S10; Experts = S13, S14, S15, S16, S17, S19 and S20); the same six novices and five experts (S14, S16, S17, S18 and S20) acted as Subjects in the Recall task.

ANALYSES: Degree of association between the chunks derived from the Recall data and the chunks from the Sorting data. Also, the degree of overlap between all the Subjects in the Sorting task with an 'ideal' expert.

1.4.3 Mental models of recursion: a programming questionnaire

A 'Program Questionnaire' was produced and Subjects were asked to indicate which of three programs contained in the questionnaire would succeed in producing a particular, required outcome. Subjects were also asked to state in their own words their reasons for selecting or rejecting each of the three programs. The novices' mental models of the behaviour of recursive procedures were determined from their selections, and from the protocols they provided on the predicted behaviour of the different programs. This experiment is reported in Chapter 4.

SUBJECTS: Nine experts (S13, S14, S15, S16, S17, S18, S19, S20 and S21) and thirty D303 Summer School students who were taking part in artificial intelligence projects volunteered to fill in and return the Questionnaire. The experts all had experience of programming in languages other than SOLO, with a minimum of a year's experience in another language.

DATA: Programs selected as correctly designed, and as incorrectly designed. Comments made by the Subjects about the behaviour of the programs.

ANALYSES: Subjects responses were categorized and the categories related to different models of recursion. Comments also were analyzed for the light they threw on the model of recursion which individual respondents possessed.

1.4.4 Perception of program structure: a transcription task

In order to determine the structure perceived by novices when they were presented with unfamiliar programs, a transcription task was designed. Subjects were required to transcribe a set of programs, each of which they were allowed to view for ten seconds on each of five trials. On each trial Subjects were required to use a different coloured pen so that the amount of information obtained and remembered on each viewing could be ascertained. What the expert sees in the first few seconds of viewing an unfamiliar program is different from what the novice sees. Generally, novices extract information from unfamiliar programs the way they would extract information from a natural language text - a line at a time. The expert, on the other hand, sees the structure of the program. But there are exceptions to both of these cases. The experiment is reported in Chapter 5

SUBJECTS: Six novice and six expert SOLO programmers were used (Novices = S5, S6, S7, S8, S9 and S10; Experts = S13, S14, S15, S16, S17, and S20).

DATA: The transcribed programs.

ANALYSES: Comparison of novice and expert SOLO programmers on number of program segments noticed; number of words recalled; number of completions.

1.4.5 The real world knowledge component of problem solving

All of the programming problems with which D303 students are confronted are couched in terms of real world events with which they are all familiar. The 'BULLETHOLE' problem (outlined and reviewed in (G), below) has elements that were designed to activate the programmer's real world knowledge, and other elements that were designed to activate programming knowledge. In order to separate out the individual effects on problem solving processes of these different sources of knowledge, the 'BULLETHOLE' problem statement was stripped of all statements that referred to programming, and the story that remained was presented to computer naive Subjects who were asked to read the story out loud, and to explain the story to the experimenter. Categories of verbal response to the story were identified and used to develop a model of problem solving for ill-defined problems. The experiment is reported in Chapter 6.

SUBJECTS: Five adult volunteers with different backgrounds (secretary, student, etc.) acted as Subjects in the experiment.

DATA: Protocols.

ANALYSIS: In-depth analysis of one of the protocols.

1.4.6 The 'pure programming' component of problem solving

1.4.6.1 An abstract programming task (1) -

In order to determine whether couching programming problems in terms of real world knowledge actually made the problem more confusing, a complement to the experiment outlined above was designed. Here, an abstract version of the 'BULLETHOLE' problem was presented. In effect, the Subjects were asked to write a program for a problem stripped of the story. No one solved the problem.

SUBJECTS: Nine D303 students volunteered to act as Subjects.

DATA: Programs designed by the Subjects.

ANALYSIS: Programs scored as correct or not correct, by the criterion that the output of the program should match the output required by the problem statement, no matter how the program achieved the required result.

1.4.6.2 An abstract programming task (2) -

The problem statement for the task outlined immediately above was considered to be very difficult to understand, and in order to circumvent any artifacts of the difficulty of the problem text, a 'clearer' version of the abstract programming problem statement was provided. The result was the same, however: no one solved the problem.

SUBJECTS: 6 D303 students volunteered to act as Subjects.

DATA: Programs designed by the Subjects.

ANALYSIS: Programs scored as correct or not correct, by the criterion that the output of the program should match the output required by the problem statement, no matter how the program achieved the required result.

1.4.7 Verbal protocols: the reading phase

An 'interactionist' theory of problem solving by computer programmers was devised from the results of the experiments reviewed in 1.4.5 and 1.4.6 above, and formalized in an Interpretation Theory for scoring verbal protocols. The theory was tested by asking several novices to solve the 'BULLETHOLE' problem. The Subjects were video-taped as they worked on the problem, and the protocols that were thus provided were used as a test of the theory. The theory has some support from the protocols analyzed as a test of the theory. The problem solving theory, the Interpretation Theory, and the analysis of the protocols are reported in Chapter 7.

SUBJECTS: Six novices (S5-S10) were video-taped as they 'thought out-loud' about the 'BULLETHOLE' programming problem and attempted to solve it.

ANALYSIS: The protocols of two of the Subjects were analyzed according to the rules of the Interpretation Theory, and the observed patterns of problem solving behaviour compared with that expected under the theory.

1.4.8 Production system models of the coding phase

In order to make completely explicit the processes that are posited to occur when novices solve programming problems, a major goal of the research reported here is to devise computer simulations of the observed behaviour. This goal has not been one hundred percent achieved, but a start has been made. In Chapter 8 of the thesis, a production system model of the coding and program evaluation phases of one of the experimental Subjects (S8) is provided. Another production system which was designed as an ideal program imitation strategy will be described, and the program imitation behaviour of another of the experimental Subjects (S5) discussed by comparison with the 'ideal' model.

Finally, the Epilogue summarizes the accomplishments arising out of the research, and also discusses the shortcomings of the research and points a direction for future work.

CHAPTER 2

'GUILT BY ASSOCIATION': ANALYSIS OF 139 PROJECT PROGRAMS

2.1 INTRODUCTION: The analyses reported in this Chapter were undertaken in order to determine the extent to which novices were able to design recursive inferencing programs after a short period of self-training in the SOLO programming language. This type of program was chosen simply because it was handy: hundreds of D303 students design such programs each academic year in partial satisfaction of course requirements. On the surface the programs that are produced show great surface variety, and another objective of the study - besides trying to determine how successful the students were - was to determine whether there was order underlying the myriad surface differences. A 'Guilt By Association' problem was originally set as an Artificial Intelligence project for D303 students in the 1980 academic year. Briefly stated, the problem involves writing a program which operates recursively to propagate a particular inference. The entire problem statement is given in Figure 2.1, at the end of this section of the chapter. Each line of the problem statement has been numbered for ease of back reference in some of the discussions to follow.

Altogether, 147 students attempted the project and designed the programs and databases which serve as the focus of studies described in this chapter. The programs and accompanying databases were collected from files kept on the University's three main DEC System-20 computers In all, 160 programs were collected, but during analyses, it was discovered that 13 of these programs were duplicates (copies of a particular student's program and database came from two different regional centres) and the copies were discarded. Of the 147 remaining programs, 8 were judged to be too long - up to 19 procedures - to try to analyze using the techniques described in this chapter. All of the 139 remaining programs and databases were subjected to analysis.

The programs and the accompanying databases show great surface diversity. The analyses described in this chapter were undertaken to determine whether, despite the surface variety, some underlying order existed.

It turns out that the databases can be broken down into a handful of basic, relational patterns. The different database structures are described in Section 2.2 below. In Section [2.3.3] the relationship between particular database structures and certain program segments will be discussed.

The programs are 'final versions', i.e., the history of their construction, and whether or not any debugging had been attempted, is unknown. Still, the programs provide us with a global perspective on novice programming behaviour. Such a wide-angle perspective is quite informative in the light it throws on the generality of the conceptual problems confronting novice programmers. The results of the analyses suggest a model of writing and debugging behaviour which will be elaborated in the discussion at the end of Section 2.3, below. Before turning to the analyses, the reader will find it interesting and instructive to read the problem statement given in Figure 2.1, and to design his own program for the stated problem.

THE 'GUILT BY ASSOCIATION'

PROBLEM STATEMENT

1) This option asks you to explore the notion of 'propagating' inferences through a database (see Units 3-4, pp78-82).

2) Suppose that SOLO had the following descriptions stored in its database:

```
LIDDY
  |
  |...ISA...BURGLAR
  |
  |...WORKSFOR...MITCHELL

MITCHELL
  |
  |...ISA...BIGLAWYER
  |
  |...WORKSFOR...NIXON

NIXON
  |
  |...ISA...PRESIDENT
```

3) Given the above descriptions, can you define a procedure called 'IMPLICATE' which makes the following inference: if someone is found to be guilty, then whoever that person works for is also guilty.

4) Here is how that procedure might work:

```
SOLO:IMPLICATE LIDDY

AHA! I'VE CAUGHT LIDDY, SO:

LIDDY
  |
    !...ISA...BURGLAR
    !
    !...WORKSFOR...MITCHELL
    !
    !...IS...GUILTY

NOW DOES LIDDY WORK FOR ANYONE?

AHA!I'VE CAUGHT MITCHELL SO:

MITCHELL
    !
    !...ISA...BIGLAWYER
    !
    !...WORKSFOR...NIXON
    !
    !...IS...GUILTY

NOW DOES MITCHELL WORK FOR ANYONE?

AHA! I'VE CAUGHT NIXON SO:

NIXON
    !
    !...ISA...PRESIDENT
    !
    !...IS...GUILTY

 NOW, DOES NIXON WORK FOR ANYONE?

 NO, SO I GUESS THAT'S ALL.
```

5) That's just a simple example. You may want to do something more elaborate-for instance, you may want to include extra CHECKS to see if other conditions are met before someone is IMPLICATEd (e.g. is that person a known criminal? etc.) You should feel free to focus on some problem other than the Watergate scandal. As you are writing your IMPLICATE procedure, you should ask yourself: do people really reason this way? If not, can you devise a better model?

FIGURE 2.1. The text of the 'Guilt By Association' problem.

2.2 DATABASE STRUCTURES

This section of the chapter provides a description of the database structures found in the databases designed by students trying to solve the 'Guilt By Association' problem. The structures are characterized in terms of the relationships that hold between nodes, and in terms of the relation names and nodes that are either shared by two or more nodes or that are unique to a single node. These notions will become clear as we proceed through the descriptions.

The structures that will be discussed here are:

1) the 'Chain Relation' List,
2) the 'One-to-many' List,
3) the 'One-to-one, Unique Successor' List,
4) the 'Many-to-one, Shared Successor' List,
5) the 'One-off Relation' List.

2.2.1 'Chain Relation' Lists

The Chain Relation, as the name suggests, links together several nodes in the database. The Chain Relation acts as a stepping stone for generating successive nodes for recursive procedures. The description of the Chain Relation structure can be formalized with the following formula:

```
a1   R   a2
a2   R   a3   => Chain Relation.
```

For example, in the following database:

```
LIDDY....ISA....BURGLAR
 |
 |.......WORKSFOR....MITCHELL

MITCHELL....ISA....BIGLAWYER
 |
 |..........WORKSFOR....NIXON

NIXON....ISA....PRESIDENT
```

given that LIDDY = a1, MITCHELL = a2, NIXON = a3, and WORKSFOR = R, a Chain Relation structure has been identified.

Of all the different database structures this is the most reliable in that it appears in almost all databases and its presence is reflected in nearly all the programs designed for the 'Guilt By Association' problem.

2.2.2 'One-to-many relation' Lists.

A One-to-many List consists in a certain relation name linking one node to two or more other nodes. The formal statement of the One-to-many List structure is:

```
        n1   R1   n2
        n1   R1   n3   =>  One-to-many.
```

In the following (partial) database, for example:

```
      BLUNT...INTIMATE...BURGESS
          |
          |.....INTIMATE...PHILBY
```

'INTIMATE' acts as a relation linking BLUNT to both BURGESS and PHILBY. The One-to-many List supports iterative processes in SOLO programming, and is seldom found in the databases designed by students working on the 'Guilt By Association' problem, although when it is part of a database design its use is reflected in 100% of the programs written to operate on these databases.

2.2.3 'One-to-one' Lists.

The One To One indicates that nodes share a particular link with other nodes, but that the second node in any triple is not shared, is unique. The formula for assigning structures to the One To One category is:

```
n1   R1   n2
n3   R1   n4.  =>  One To One.
```

An example database containing two instances of One To One is given in the following design:

```
   FRED...ISA...BURGLAR
    ¦
    ¦.....DRINKS...BEER
    ¦
    ¦.....WORKSFOR...JOHN

   JOHN...ISA...FENCE
    ¦
  ¦.....DRINKS...WHISKY
    ¦
    ¦.....WORKSFOR...BOB

 BOB....ISA...RACKETEER
    ¦
    ¦.....DRINKS...CHAMPAGNE
```

Here, ISA links different nodes to unique successor nodes (BURGLAR, FENCE, RACKETEER). The same, of course, applies to DRINKS.

2.2.4 'Many-to-one' relation Lists

The 'Many-to-one' relation is the opposite of the 'One-to-many relation' List. Here, different nodes share both a relation name and the second node in an associative triple. The formula for the

Many-to-one structure is the following:

```
n1   R1   n2
n3   R1   n2. => Many-to-one
```

Here is an example database containing a double Many-to-one List structure: 'ISA SPY' and 'WASAT CAMBRIDGE'.

```
BLUNT...ISA...SPY
    |
    |....WASAT   CAMBRIDGE

  PHILBY...ISA...SPY
    |
    |....WASAT...CAMBRIDGE

  BURGESS...ISA..SPY
    |
    |....WASAT...CAMBRIDGE
```

It should be noted that not all nodes in the database need be involved when a Many-to-one List is constructed. The following database also yields this construction:

```
LIDDY...ISA...BURGLAR
 |
 |......WORKSFOR...MITCHELL

MITCHELL...ISA...BIGLAWYER
 |
 |......KNOWS...PROPOSITION1
 |
 |......WORKSFOR...NIXON

NIXON...ISA...PRESIDENT
 |
 |......KNOWS...PROPOSITION1
```

where only MITCHELL and NIXON are linked by the shared relation, 'KNOWS', to the successor node 'PROPOSITION1'.

2.2.5 'One-off Relation' Lists

The One-off Relation is a triple in which the relation name is not shared with any other triple in the database. The formula for assigning triples to this structure is:

```
n1   R1   n2
  n2   R2   n3    => One-off Relation.
```

An example of a database containing several such structures is:

```
BOB...ISA...MAN
   !
   !....LOVES...MARY

  MARY...ISA...WOMAN
   !
   !....DRINKS...METHS
   !
   !....DESPISES...ADRIAN

ADRIAN...ISA...MAN
 !
   !....OWNS....FIDO
```

where LOVES, DRINKS, DESPISES and OWNS are all One-off Relation structures. That is, the One-off Relation linking 'MARY' and 'METHS' is 'DRINKS' and it is the only example of the relation name anywhere in this particular database.

2.3 THE STRUCTURAL DESCRIPTIONS OF PROGRAMS

A program can be defined as some number of program segments concatenated in different ways, by different students, depending upon the intentions and the skills of the student designing and coding the program. The structure of a program is described by assigning labels to program segments. Not all segments are of interest, so not all segments are labelled. For example, printed strings, although they are oftentimes quite informative of a programmer's intentions, are not labelled in the descriptions provided. The analysis was designed to determine whether or not side-effects to the database occur in a given program, and the conditions under which the side-effect is made. The labels, and the conditions under which they are assigned, are:

1) Unconditional Side-Effect. If the first significant program event (disregarding printed strings) is a side-effect to the database the segment is labelled as a USE (Unconditional Side-Effect) segment. Thus, if a program begins:

1) NOTE /X/ IS GUILTY then the USE label is applied. In the example programs below, the label can be found on the right hand side of the arrow pointing from the particular segment, e.g.,

```
1) NOTE /X/ IS GUILTY ----------> ;  USE.
```

2) Conditional Side-Effect. If the first two significant program events are a conditional statement, and a side-effect segment conditional on the outcome of the conditional test, then the segments are treated as a single CSE (Conditional Side-Effect) segment. There are different ways that these segments can be designed, and these are two of them:

```
A)  1 CHECK /X/ ISA BURGLAR -----------------------|
                                                   |---> ; CSE
    1A If Present: NOTE /X/ IS GUILTY; CONTINUE ---|

B)  1 CHECK /X/ ISA BURGLAR -----------------|
    1A If Present: CONTINUE                  |-----> ; CSE
    1B If Absent:  EXIT                      |
2 NOTE /X/ IS GUILTY -------------------|
```

3) Generate Next Object and Self. The standard recursive segment, (described in Chapter 1) is labelled as a GNO/S segment, as follows

```
CHECK /X/ <relation> ?   -----------------------|
                                                |---> ; GNO/S
If Present <procedure-name>  * ; EXIT ----------|
```

4) Conditional on Star, and Conditional on X.

There are instances where another program segment intervenes between the combined segments of the GNO/S construction. In the next example, a conditional segment occurs after the GNO segment and before the recursive call to the procedure (the Self segment). The conditional uses the value of the wild-card variable in the first node slot, so the label applied is COND-* (Conditional on Star).

```
CHECK /X/ WORKSFOR ? ----------------------->  ; GNO
If Present: CONTINUE
If Absent:  EXIT
CHECK *  EXCESS  INCOME  -------------------->  ; COND-*

If Present: IMPLICATE * ; EXIT ------------->  ; SELF
If Absent:  EXIT
```

Sometimes the conditional statement uses both 'X' and '*' so I assume that 'X' has precedence. In such cases the conditional statement is labelled COND-X (Conditional on X). Here is an example:

```
CHECK /X/ WORKSFOR ? ----------------------->  ; GNO
If Present: CONTINUE
If Absent:  EXIT
CHECK /X/ KNOWS  *  ------------------------>  ; COND-X

If Present: IMPLICATE * ; EXIT ------------->  ; SELF
If Absent:  EXIT
```

Program descriptions were used to derive the designer's model of the problem. That is, if a program begins with an Unconditional Side-Effect, such as 'NOTE /X/ IS GUILTY' it has been assumed that the programmer interpreted the problem statement (see line 3 of the problem text, Figure 2.1) to mean that the guilt of some person (for example, LIDDY) can be assumed from the start. The analysis indicates that students have their own stylized interpretations, or mental models, of the task at hand. The observed program structures ought to correspond to students' mental models of the task. Some of these mental models are 'appropriate', in that they address the problem as stated, while others introduce certain anomalies which preclude a satisfactory solution. Such 'inappropriate' models could actually be artifacts of students 'thinking in SOLO' and getting led astray. Although the students have a great deal of freedom to choose ways of implementing solutions, they typically resort to a few common approaches. In the rest of this section of the chapter, structural descriptions of five different program designs will be provided. Each of these designs was implemented by at least ten students attempting the 'Guilt By Association' problem. These five types account for 65% of all the programs.

The account will be straightforward and follow this format. Each subsection begins with the name of a derived problem model - the model presumed to exist in the programmer's head and to be accurately reflected in the program produced. An example program will be provided, and the derived model of reasoning formulated from the structural description. Finally, the number of programs of that particular design will be provided.

2.3.1 The 'Convict Everybody' Problem Model

The program in Figure 2.2 was written by one of the project students, and consists of two segments, USE and GNO/S.

```
TO IMPLICATE /X/
1 NOTE /X/ IS GUILTY       ------------->  ; Unconditional Side-effect
                                             segment = CONVICT FIRST

2 CHECK /X/ WORKSFOR ?      ---------|     ; Generate Next Object Segment
                                     |-->  ;    and
2A If Present: IMPLICATE *; EXIT --|         Self Segment = CONVICT THE
                                                            REST
2B If Absent: EXIT
DONE
```

FIGURE 2.2. The 'Convict Everybody' program.

The first segment is an unconditional side-effect, and this is followed by a standard recursion segment. The name of the problem model the student is presumed to have in his head when he designs such a program is called the 'Convict everybody' model, which may be summarized as something like this:

1) The guilt of some person P is assumed, as given (1st program segment) and,

2) other persons P' are guilty by dint of standing in some relation to P (in this instance WORKSFOR) or to each other (2nd program segment).

Thirty (21.6%) out of the 139 programs (see Table 2.1) were of this design, the most frequent design found amongst these programs.

2.3.2 The 'Convict First Person, Prosecute Rest' Problem Model

There are two versions of this problem model, one derived from program designs containing a COND-* segment, the other derived from those containing a COND-X segment.

A program featuring COND-* is given in Figure 2.3.

```
TO IMPLICATE /X/
1 NOTE /X/  IS  GUILTY  ---------->  ; Unconditional Side-Effect segment

2 CHECK /X/  WORKSFOR  ?  -------->  ; Generate Next Object segment
2A If Present: ...
2B If Absent: ...
3 CHECK *  HAS  EXCESSINCOME ----->  ; Conditional On Star segment

3A If Present: IMPLICATE *; EXIT  ---------->  ; Self segment
3B If Absent: EXIT
DONE
```

FIGURE 2.3 The 'Convict First Person, Prosecute Rest' program 1.

The structure of the program is this. There is an Unconditional Side-Effect at the first segment. Following this, the Generate Next Object occurs, and is separated from the Self segment by a COND-* segment.

The problem model presumed to exist in the program designer's head is something like this:

1) The guilt of some person P is assumed, is given (1st program segment) and,

2) other persons P' are guilty by dint of standing in some relation to P (in this instance WORKSFOR) or to each other (2nd program segment)

IF there is also evidence of their guilt (such as 'HAS EXCESSINCOME')x(3rd program segment).

Table 2.1 shows that this type of program was produced by 16 students, which represents 11.5% of the total number of programs.

The programs featuring a COND-X segment have more or less the same structural description. A difference is that in the COND-X type programs, the conditional segment can be placed before the GNO/S segment or may intervene between the GNO and Self segments. This is not possible with the COND-* type programs, of course, because a value of the wild-card variable must be generated before it can be used. An example program is given in Figure 2.4.

```
TO IMPLICATE /X/
1 PRINT "I'VE CAUGHT" /X/ "SO"
2 NOTE /X/ IS GUILTY  --------------------->  ; USE
3 CHECK /X/ HAS LOTSOFMONEY  -------------->  ; COND-X
3A If Present: CONTINUE
3B If Absent:  EXIT
4 CHECK /X/ WORKSFOR ?  --------------!
                                      !---->  ; GNO/S
4A If Present: IMPLICATE *; EXIT -----!
4B If Absent:  EXIT
```

FIGURE 2.4 The 'Convict First Person, Prosecute Rest' program 2.

The problem model derived from this structure is:

1) The guilt of some person P is assumed, is given (1st program segment) and,

2) other persons P' are guilty by dint of standing in some relation to P (in this instance WORKSFOR) or to each other (3rd program segment)

IF also a particular description (in this case 'HAS LOTSOFMONEY') is applicable to P.

Thirteen (9.4%) of all programs exhibited the COND-X feature.

2.3.3 The 'Prosecute Everybody' Problem Model

An example of the program design implementing this problem model is given in Figure 2.5.

```
TO IMPLICATE /X/
1 CHECK /X/ HAS RECORD  -----------------------> ; Conditional segment
1A If Present: NOTE /X/ IS GUILTY; CONTINUE ---> ; Side-Effect segment
1B If Absent:  EXIT
2 CHECK /X/ PAIDBY ?  -------------------------> ; Generate Next Object
2A If Present: IMPLICATE * ; EXIT  ------------> ; Self segment

2B If Absent: PRINT "NO MATE IT'S A FAIR COP!"; EXIT
DONE
```

FIGURE 2.5 The'Prosecute Everybody' program.

The structure of the program in Figure 2.5 is a conditional side-effect first segment, and the second segment is a standard recursion segment.

The problem model presumed to exist in the mind of this student would be something like this:

1) Some person P is guilty only if there is evidence of his guilt (e.g., 'P' HAS RECORD), and

2) other persons P' are guilty by dint of their standing in some

relation to P (e.g., P is PAIDBY P') or to each other, IF there also exists the same evidence against P' as against P.

Table 2.1 shows that 15 of the 139 programs were of this type: 10.8% of the total.

2.3.4 The 'Prosecute First, Convict Rest'. Problem Model

An example implementation of this problem model is given in Figure 2.6.

```
TO GUILTTEST /X/
1 CHECK /X/ ISA BURGLAR                       ; FILTER procedure
 1A If Present: IMPLICATE /X/ EXIT
 1B If Absent: EXIT
DONE

TO IMPLICATE /X/
1 NOTE /X/ IS GUILTY -------------> ; Unconditional Side-Effect segment

2 CHECK /X/ WORKSFOR ? -----------> ; Generate Next Object segment

 2A If Present: IMPLICATE *; EXIT --> ; Self segment
 2B If Absent: EXIT
DONE
```

FIGURE 2.6 The Prosecute First, Convict Rest' program

The program consists of a main procedure, 'GUILTTEST' (a FILTER) containing a Conditional segment which tests for the presence of a particular description of the parameter node (ISA BURGLAR, in this case), and, dependent upon the presence of the pattern, triggers the subprocedure, 'IMPLICATE'. The subprocedure contains an Unconditional Side-Effect first segment, and the standard Generate Next Object and Self segments in steps 2 and 2A. The FILTER functions to remove the Conditional pattern '/X/ ISA BURGLAR' from the operation of the recursive procedure, 'IMPLICATE'.

The problem model derived from this design is:

1) Some person P is guilty only if there is evidence of his guilt (e.g., 'P ISA BURGLAR'), and

2) other persons P' are guilty by dint of their standing in some relation to P (e.g., WORKSFOR) or to each other.

Table 2.1 shows that only 13 such programs were designed, just 9.4% of the total.

SUMMARY and DISCUSSION: Nearly two thirds of the project students (63%) designed programs of one or another of the five types described above. Altogether, 87 programs fell into these categories. Table 2.1 summarizes the data. The first column of the table gives the structural description of the five different types of program, and the second column gives the number of programs of each type, (and the percentage of the total of 139 programs that these figures represent)

	TYPE	TOTAL
The 'Convict everybody' model	USE; GNO/S	30 (21.6%)

The 'Convict first person, prosecute rest' model	USE;
	GNO;
	COND-*;
	SELF
		16
		(11.5%)

The 'Prosecute everybody' model	CSE; GNO/S	15 (10.8%)
The 'Prosecute first person, convict rest' model	FILTER	13 (9.4%

The 'Convict first person, prosecute rest' model	USE;
	GNO;
	COND-X;
	SELF
		13
		(9.4%)

TABLE 2.1

Of the remaining 52 programs, 18 are of interest because of the light they throw on a particular, widespread conceptual problem with which students are confronted (or so it will be argued below) when trying to design a program for the 'Guilt By Association' problem. These programs have been labelled 'Hand Operated Recursion', 'Flattened recursion', and 'Enablement pattern' programs. They will be discussed further below. The remaining programs consisted of either: a Generate Next Object and Self segment, and nothing more; a Conditional or Unconditional side-effect segment, and nothing more; a combination of Unconditional and Conditional side-effect segments, and nothing more; and so on. These 52 programs tell us that about one third of the students encountered a lot of difficulty in writing recursive procedures. In a few cases (14 programs) there were signs that designing any program would have been beyond some of the students.

During the analysis of the programs, it was noticed that most had bugs of one sort or another, but these were not classified and counted so no statistics are available. There were two reasons for this. First, a detailed analysis of errors that occur in novice SOLO programs has been performed by Lewis (Lewis, 1980). Second, many of the types of errors that were noticed have been analyzed and discussed in terms of a debugging assistant for novices in SOLO programming already (Hasemer, in press).

The frequency of occurence of particular database structures accompanying programs of the five program types described above and summarized in Table 2.1, is given in Table 2.2. The Table shows the number of programs (1st column), and the program structural descriptions (2nd column). In subsequent columns the number of databases containing a particular structure and the number of programs which reflect (use) the particular structure are indicated. For example, in the third column, the Chain Relation (CR) occurs in all thirty databases which accompanied the USE + GNO/S programs, and all thirty programs had a segment which operated on the Chain Relation (30[30]). The Chain relation also occurs in all 13 of the databases accompanying the FILTER programs, but only 12 of the programs have a segment which operates on the Chain Relation (13[12]).

			CR	Many-1	1-1	OneOff	1-Many
1)	30	USE; GNOS	30 [30]	7 [3]	26 [3]	19 [4]	0 [0]

2)	16	USE;
		GNO;
		COND-*;
		SELF
			15
			[15]
				10
				[8]
					9
					[4]
						12
						[7]
							2
							[2]

3)	15	CSE; GNOS	14 [14]	13 [13]	9 [3]	8 [3]	3 [4]
4)	13	FILTER	13 [12]	4 [0]	9 [7]	11 [9]	3 [1]

5)	13	USE;
		GNO;
		COND-X;
		SELF
			12
			[12]
				9
				[9]
					9
					[4]
						8
						[3]
							0
							[0]

TABLE 2.2

The question to be raised at this point is this. Do the 87 programs which are accounted for by the designs given in Table 2.2 accurately reflect their designers problem models? That is, do the problem models that were derived from the structural descriptions of the programs accurately reflect the models the students had in mind when they started writing code?

The problem model which the author of the problem statement intended that the student should abstract (Eisenstadt; personal communication) is graphically represented in Figure 2.7

```
                 worksfor                         worksfor
    LIDDY-------------------->MITCHELL-------------------->NIXON
      |                          |                           |
   is |                       is |                        is |
      |                          |                           |
      |                          |                           |
      |------> GUILTY  <-------------------------------------|
```

FIGURE 2.7 A graphic representation of the 'Convict everybody'

problem model.

The problem was meant to be isomorphic with the INFECT problem given in the Project Manual, page 80. Note that the first line of the problem statement contains a pointer to the INFECT program, and it was expected that students would use that procedure as a model for writing their program for this problem.

However, some readers might well be expected to interpret 'if someone is found to be guilty' to mean that if evidence of 'someone's' criminality can be discovered, then that 'someone' can be said to have been 'found out', 'found' guilty. Support for this interpretation, in the reader's mind, would be the information in the problem's example database: LIDDY ISA BURGLAR. A graphic description of the problem model reflected in the FILTER programs is given in Figure 2.8 (More will be said about the Figure 2.8 representation below.)

```
                  worksfor                           worksfor
     LIDDY-------------------->MITCHELL-------------------->NIXON
      |  |                        |                           |
      |  |                        |                           |
      |  | isa                    |                           |
      |  |                        |                           |
      |  |--BURGLAR               |                           |
      |                           |                           |
 is   |                    is     |                           | is
      |                           |                           |
      |------>GUILTY    <---------| <-------------------------|
```

FIGURE 2.8. A graphic representation of the 'Prosecute first, convict others' problem model as reflected in the FILTER programs.

In all, 30 students designed programs which perfectly reflect the 'Convict everybody' problem model (the first row in Table 2.2). The program designs in rows 2 and 5 of Table 2.2 (giving a further 29 programs) are elaborations of the 'Convict everybody' model. These elaborations are suggested to the project students in the fifth statement of the problem text, where it is suggested that students may want to "do something more elaborate, such as include extra Checks, etc."

The 13 FILTER type programs reflect the 'Prosecute first, convict others' problem model. Thus, on the surface, indications are that the majority of project students abstracted the 'Convict everybody' model from the problem text.

The strong intuition is that the figures should be reversed, that many more of the students should have abstracted the more sophisticated model from the problem statement. In attributing problem models to students on the basis of the structural descriptions of their program designs we may be disguising important problem solving processes. The implication of assigning such models is that the students have no difficulty translating their mental model of the problem in hand into program code. Unfortunately, the data is 'silent'. We have the programs but not the programmers. We can raise certain questions about the programs, but their designers are not available to say "No, that's not what I meant at all!", or to agree with us, as the case may be. In order to discover which questions might be asked of the data, a volunteer Subject, S1, was given the problem to solve. S1 was a volunteer, a student taking the second level course 'Introduction to psychology' at the Open University. In order to make the experiment as valid as possible for comparison with the behaviour of the project students, S1 was treated like any other D303 student. S1 read the early

'Cognitive Psychology' course material in her own home, at her own pace. She studied the Programming Manual in the same circumstances, and performed all the study centre activities which are required of all D303 students. Only then was S1 given the project material, and, like all other students, S1 was advised to try to solve the problem with pencil and paper at home in order to save time at a terminal site. Students have to book a terminal at one of the Open University's numerous study centres, and each student is allowed to book no more than one hour's computing time in any one day. Most students have to travel some distance to their local study centre. If the student designs a program before going to the study centre all he has to do when he gets there is type it in and test it. If there is a bug in the program he can spend most of his hour repairing the program. When S1 had attempted the project at home, she was brought into our laboratory and provided with a terminal to run, test and, if necessary, debug her program. In fact, S1 was quite discouraged after having spent some considerable time designing different programs for the problem. The transcript of part of the first tape recorded session with S1 is provided in Appendix A. This first part of the protocol is a retrospective account by S1 of the various attempts that were made at solving the problem.

In all, S1 managed four different designs at home. These are given in Figures 2.9 - 2.12. Three of these programs (Figures 2.9, 2.10, and 2.11) have the same structural descriptions as programs already discussed, and the fourth (Figure 2.12), while not discussed previously, has also been found amongst the 139 project programs.

```
'Unreachable Code'
TO IMPLICATE /X/
1 CHECK /X/ ISA BURGLAR
```

```
1A If Present: NOTE /X/ ISA CROOK; CONTINUE
1B If Absent:  PRINT /X/ "IS IN THE CLEAR"; EXIT
2 CHECK /X/ WORKSFOR ?
2A If Present: IMPLICATE *
2B If Absent: EXIT
DONE
```

FIGURE 2.9. The 'Unreachable Code' program designed by Subject S1.

```
'FILTER'

TO IMPLICATE /X/
1 CHECK /X/ IS A BURGLAR
1A IF PRESENT NOTE /X/ IS GUILTY; CONTINUE
1B IF ABSENT: PRINT X IS NO CRIMINAL; EXIT

TO GUILTTEST /X/
1 CHECK /X/ WORKSFOR ?
2A IF PRESENT: NOTE "* IS A CROOK"; EXITr
2B IF ABSENT: PRINT "* IS IN THE CLEAR"; EXIT
```

FIGURE 2.10. An unsuccessful attempt by Subject S1 to design a program for the 'Prosecute first, convict others' problem model.

```
'Flattened recursion'

CRIMECHECK
1 CHECK X ISA BURGLAR
P: PRINT /X/ IS GUILTY CONTINUE
A: EXIT
2 CHECK X WORKSFOR ?
P: IMPLICATE *; CONTINUE
A: EXIT
3 CHECK * WORKSFOR ?
P: IMPLICATE *A; CONTINUE
A: EXIT
4 PRINT ALL GUILTY
```

FIGURE 2.11 An attempt to 'flatten out' the recursive process by Subject S1.

```
'USE; GNO/S'

TO IMPLICATE /X/
 1 NOTE /X/ IS GUILTY
 2 CHECK /X/ WORKSFOR ?
 2A If Present: IMPLICATE *; EXIT
 2B If Absent:  EXIT
 DONE
```

FIGURE 2.12 The program finally produced by Subject S1, reflecting the 'Convict everybody' problem model.

The problem with the 'Unreachable Code' in Figure 2.9 is that the conditional statement at step 1 of the procedure causes early termination of the program. The conditional statement eventuates in 1) the side-effect on the parameter node at step 1A, and 2) continuation to step 2 of the program. Since the only node having the description 'ISA BURGLAR' is LIDDY, the program terminates before 'MITCHELL' and 'NIXON' can be labelled guilty. As may be seen in the protocol in Appendix 2, the strategy used by S1 in designing the 'Unreachable Code' program was to break the problem down into parts and to write code for each of the parts separately. Each worked on its own, but when the parts were put together there occured the 'clobber' interactions described by Sussman (1970), or the non-linear main-step failure described by Goldstein (1975). There are several indications in the protocol that the Subject prefers to work at the individual segment level of programs - achieving first this in a segment of code, then something else.

The structural description of the 'Unreachable Code' program is the same as the programs in the 3rd row of Table 2.2: 'CSE; GNO/S'. The difference between the two types of program is that the 'CSE; GNO/S' programs succeed. The argument is that the students who designed the 'CSE; GNO/S' programs had abstracted the 'Prosecute first, convict others' model from the problem statement, but did not 'get the idea' of using a FILTER to separate the conditional pattern, which is required of the 'Prosecute first, convict others' model, from the recursive call to the procedure. (See S1's protocol in Appendix 2.) A compromise solution would be to create a 'stereotype' for database nodes, and to use this stereotype as a sort of 'database implemented' FILTER. The stereotype is realized by devising Many-to-one Lists in the database. The database designed for the 'CSE; GNO/S' program described in Section 2.3.2, above, was this:

```
JOE....ISA....HITMAN
 !
 !.....HAS....RECORD
 !
 !.....PAIDBY....ANGELO
ANGELO....ISA....FIXER
 !
 !........HAS....RECORD
 !
 !........PAIDBY....GODFATHER
GODFATHER....ISA....BOSS
 !
 !...........HAS RECORD
```

Each of the database nodes has the relevant descriptor 'HAS RECORD', a device which permits all the nodes to pass the conditional test which is the first segment of the type of program under discussion. The device prevents the early termination of programs that have a conditional first segment.

Table 2.2 also contains information on the number of databases containing the Many-to-one structure for all the programs of each program type. For example, of the 30 databases designed for the 30 'USE; GNO/S' programs (first row in Table 2.3), 7 contain the Many-to-one List structure. Of these 7 occurrences, 3 are actually reflected in the 'USE; GNO/S' programs. That is, some of the 'USE; GNO/S' programs could have been classified as 'CSE; GNO/S' designs. The reason they weren't is because a different solution to the early termination problem is used by some students: the 'Continue Regardless' solution. The solution involves writing a conditional first segment, and then writing 'CONTINUE' instructions in the flow of control slot on both branches of the conditional statement, e.g.,

```
CHECK /X/  ISA  BURGLAR
1A If Present: CONTINUE
1B If Absent:  CONTINUE
2 NOTE /X/  ISA  CROOK
2A If Present: ...
```

FIGURE 2.13

Whenever this construction was encountered, it was considered that, since the conditional statement really had no effect on subsequent side-effect segments, the side-effect would be labelled as 'Unconditional'.

Of special interest to us at the moment is the number of FILTER programs in which the Many-to-one List was incorporated in the accompanying database - 4. Of these, the structure is reflected in none of the FILTER segments of the programs produced. The figures suggest that nearly a third of the students who produced FILTER programs toyed with the idea 'stereotyping' solution, before hitting on the idea of using a FILTER.

Table 2.3 shows that the Many-to-one structure occurred in the databases of only 50% of the students who designed the programs of the type 'USE; GNO; COND-*; SELF', and of 69% of those who designed 'USE; GNO; COND-X; SELF' programs. Since both these types combine a conditional statement with a recursive segment, it might be expected that nearly all of the databases would contain the 'stereotype' structure. However, many of these particular program designs use a disjunction of conditions in combination with the recursive segment to trigger the recursion. That is, the trigger for recursion might be:

'/X/ WORKSFOR *' and '* EXCESS INCOME', OR, '/X/ WORKSFOR *' and '* KNOWS PROP1'. Such designs typically use the 'One-off' or the 'One-to-one' relation lists.

PROGRAM DESIGNS	MANY-1 RELATION
USE; GNO/S	3/30 = 10%
USE; GNO; COND-*; SELF	8/16 = 50%
FILTER	0/13 = 0%
CSE; GNO/S	14/15 = 93.3%
USE; GNO; COND-X; SELF	9/13 = 69.2%

TABLE 2.3

S1 did get the idea of using a FILTER procedure (Figure 2.10) to overcome the early termination problem, but the program failed because S1 could not figure out how to trigger the subprocedure. (Of the 13 FILTER programs produced by the project students, 1 was like this.) The third solution tried by S1 (Figure 2.11) represents a kind of 'flattened out' recursion process (see below). In this program, the Subject uses two 'GNO/S' segments to generate and recurse on the values of 'MITCHELL' and 'NIXON'. We can compare this with one of the 'Flattened recursion' programs found in the programs in the 'OTHER' category mentioned above.

```
TO IMPLICATE /X/
1 CHECK /X/  ISA  THIEF            ---------------> ; Conditional
 IP: PRINT "I'VE CAUGHT" /X/; CONTINUE
 IA: PRINT /X/ "DOES NOT STEAL AND THEREFORE IS INNOCENT"; EXIT

2 CHECK /X/  WORKSFOR  ?A          ---------------> ; GNO
 IP: PRINT "AHA! I'VE ALSO CAUGHT" *A; CONTINUE
 IA: PRINT /X/ "DID IT ON HIS OWN"; EXIT

3 NOTE /X/  IS  GUILTY             ---------------> ; Side-effect (X)
```

```
4 NOTE *A  ISALSO  GUILTY           ---------------> ; Side-effect (*A)

5 CHECK *A  WORKSFOR  ?B            ---------------> ; GNO
 IP: PRINT "AHA! I'VE CAUGHT" *B "AS WELL"; CONTINUE
 IA: PRINT /X/ "AND" *A "WERE IN IT ALONE"; EXIT

6 NOTE *B  IS  GUILTY               ---------------> ; Side-effect (*B)
7 PRINT "NOW, DOES" *B "WORK FOR ANYONE?"

8 CHECK *B  WORKSFOR  ?C            ---------------> ; GNO
 IP: PRINT "IT MUST BE MY DAY, I'VE EVEN CAUGHT" *C; CONTINUE
 IA: PRINT *B "MUST BE THE BOSS, SO THAT'S ALL"; EXIT
9 PRINT *C "IS GUILTY ALONG WITH THE OTHERS"

10 CHECK *C  WORKSFOR  ?D           ---------------> ; GNO
 IP PRINT "I'M HANDING OVER THE CASE TO SCOTLAND YARD NOW"; EXIT
 IA PRINT *C "IS THE BIG BOSS SO I GUESS THAT'S ALL"; CONTINUE
DONE
```

FIGURE 2.14

In this program, the 'recursion' has been completely flattened out. What this means is that rather than include a recursive call to the procedure, the designer builds the required actions - the side-effects which would normally be consequents of the recursion - into subsequent lines of code. The flattened out process overcomes the early termination problem with programs having a conditional first segment but no stereotype database to operate on.

In the end S1 opted to simplify the problem model (although she thought it was cheating and was reluctant to do so - see Appendix 2.) The model which she was capable of designing a database for was the 'Convict everybody' model.

2.4 IMPLICATIONS: PROBLEM UNDERSTANDING

The major implication of these findings is that the problem understanding stage of problem solving in programming is critically important. Whether or not a program is correctly designed depends upon the intentions of the person doing the programming. Some students seemed to read into the 'Guilt By Association' problem statement a 'Convict Everybody' model of the problem to be solved, while others seemed to read into it a 'Prosecute first;: convict others' model of the problem. The finding underlines not only the importance of the problem understanding phase of problem solving, but also the importance of the a priori models which the problem solver brings to bear during this stage of problem solving.

To date, very little research has been attempted on the problem understanding phase in computer programming, although both Brooks and Schneiderman have commented on this phase of (expert) programming activity.

Brooks (1976) has partitioned programming activity into three distinct processes - Problem Understanding, Method Finding, and Coding, each of which is claimed to follow the other. Although Brooks has little to say about the important processes of Problem Understanding and Method Finding he suggests that the 'UNDERSTAND' program (Hayes & Simon, 1974) is a more or less adequate model of the problem understanding process.

In 'UNDERSTAND', problem understanding involves two subprocesses, Language Understanding and Model Construction. The product of these processes is a problem space containing the initial and goal states of a problem, the problem objects and their properties and relations, and, finally, the operators for transforming the initial into the goal state, plus any restrictions on the use of the operators. This final representation can then be operated on by a special purpose problem solving mechanism. If the problem is not solved in this 'Solution' phase, the problem understanding mechanisms are instantiated a second time and the process begins again.

The UNDERSTAND program operates on 'well defined' problems, such as the Tower of Hanoi problem, or the Tea Ceremony problem. These are problems that specify all the information a problem solver needs in order to solve the problem. A drawback of the 'UNDERSTAND' model as a likely 'front end' for Brooks' model of programming activity is that programming problems are not well defined, in the sense that 'operators' are not usually explicitly indicated in programming problem statements. Programmers have to recognize which operators are relevant from an analysis of various features of the problem statement; that is, selection of operators is indirect. If the problem does not cue a known algorithm, then the programmer must devise one. Brooks (1977) makes much the same observation and introduces the notion of 'method finding' to account for the programmer's need to determine a suitable algorithm, but he has nothing to say about method finding processes other than to indicate reflections of such processes in the protocols he analyzed. Part of the goal of this thesis is to demonstrate such processes in operation.

Equally important with respect to 'UNDERSTAND', solution processes do not begin operating until enough is Known about the problem to get the processes going. Unfortunately, this 'enough' involves Knowing everything. In 'well defined' problems solution processes cannot get started until the operators (and the restrictions on their use) are Known, and in the Tea Ceremony problem, which is used by Hayes & Simon to exemplify the operation of the system, this information is not finally given until the penultimate line of the problem statement. Thus, problem understanding processes necessarily precede any attempts at a solution to the problem.

A similar model of expert programming behaviour has recently been proposed by Shneiderman (1980). It is called a 'Semantic/Syntactic' model and characterizes programming activity in terms of a problem understanding phase, followed by a planning phase. In the understanding phase the problem is represented in working memory in terms of initial and goal states. In the planning phase general approaches to the problem are generated (i.e., semantic Knowledge is brought to bear) and these approaches are subsequently refined and code is generated (i.e., syntactic Knowledge is applied). The important resemblance to Brooks' model is that planning Knowledge (= method finding) is brought to bear only after a representation of the problem has been constructed in working memory.

In summary, both Brooks and Shneiderman separate the processes of problem understanding, and subsequent problem solving processes, and both discuss only programming by experts. Interestingly, neither of these researchers mention a role for Knowledge other than domain knowledge in their theories of problem solving by computer programmers.

In Chapter 7, a model of problem solving behaviour applicable to both expert and novice computer programmers will be presented. In contrast to the theories described above, the model to be presented posits that solution processes come into operation at the same time as problem understanding processes. That is, understanding and solution processes are presumed to co-occur.

2.5 TRANSLATION BETWEEN PROBLEM AND PROGRAM REPRESENTATIONS

The findings suggest that a major difference between novice and expert computer programmers is the translation from a completed problem representation into code. For instance, Brooks, in his massive study of the coding behaviour of an expert programmer, suggests that, given an adequate representation of the problem, plus a bridge, or method for translating the representation into a programming language, coding itself is fairly non-problematic. Most of the hard work during this phase of programming has to do with updating the contents of Long Term Memory so that the programmer can keep track of variable names, and so forth. In contrast, the evidence reviewed above suggests that novices may have considerable difficulty in designing a program which accurately reflects the problem model they have in their minds. The reasons for this may be various, and presumably are different for different programmers.

One possibility is that novices, having little programming experience, have internalized very few ideal programs which they can draw upon in designing new programs. Or it may be that although they have internalized an adequate store of such programs - adequate, that is, to solve a given problem - access to the appropriate knowledge is not driven by elements of the problem statement. Or it may be that the novice does not think to combine knowledge fragments. For example, consider a novice who can write a program involving subprocedures, because he remembers examples from the Programming Manual. He also knows how to write recursive procedures. But given an adequate representation of the problem, he fails to find an ideal solution because it does not occur to him to combine what he knows about recursion with what he knows about using subprocedures. Schoenfeld (1980) has shown that college students can be shown to have knowledge necessary to solve mathematics problems, and yet often fail to use that knowledge to solve problems.

It is also possible that novices simply fail to achieve an adequate representation of the problem, a representation on which a satisfactory solution could be based. They may fail to achieve such a representation because they think in terms of the programming language they are using. Odd elements of programming knowledge may contribute to the construction of a problem representation which then is resistant to change. Let me illustrate. In the discussion above, the following representation of the 'Prosecute first, convict others' problem model was given.

```
                  worksfor                          worksfor
LIDDY-------------------->MITCHELL------------------->NIXON
 |  |                       |                           |
 |  |                       |                           |
 |  | isa                   |                                 |
 |  |                       |                           |
    |  |--BURGLAR               |                               |
    |                           |                         |
is  |                       is  |                         | is
    |                           |                         |
    |------>GUILTY     <--------| <-----------------------|
```

FIGURE 2.15

The representation has a strong quality of closure about it, suggesting something whole, complete in itself. It suggests a program which should be realized in a single procedure. D303 students are taught to write recursive procedures to propagate 'an' inference, such as '/X/ HAS FLU'. The problem involves the inference '/X/ IS GUILTY'. The problem statement points to an example program that might be imitated, and the example program does not contain a subprocedure. The solver actually wants to create a program that makes two inferences, one that Liddy is guilty because he is a burglar, and the other that associates of Liddy (and of Liddy's associates) are guilty by reason of their relationship with one another. But he has represented the problem as one in which only a single inference is required. and Nixon are guilty.

Arguably, a better representation of what the programmer wants to achieve, given the 'Prosecute first, convict others' model, is this:

```
               worksfor                    worksfor
   LIDDY-------------------->MITCHELL------------------->NIXON
    |  |                        |                         |
    |  |                        |                         |
    |  | isa                    |                               |
    |  |                        |                         |
    |  |--BURGLAR               |                               |
    |                           |                         |
is  |                       is  |                         | is
    |                           |                         |
    |------>GUILTY              |------->IMPLICATED<------|
```

FIGURE 2.16

The interpretation offered for the data presented in this Chapter argues that novices are capable of re-representing a problem once they run into difficulties, but that an optimal representation is not achieved. Faced with the 'early termination' problem, the project students seem to identify the 'bug' as 'non-equivalence of patterns': Liddy is a burglar, but Mitchell and Nixon aren't. If they were all 'the same' the early termination problem could be overcome. A representation of the 'stereotype model is this:

```
                 worksfor                        worksfor
      X------------------------->Y----------------------->Z
      !                          !                        !
      !                          !                        !
      !                          !                        !     !
  isa !                      isa !                        ! isa
      !                          !                        !
      !--------------------->WHATEVER<--------------------!
      !                          !                        !
      !                          !                        !
      !                          !                        !
      !------------------------>GUILTY<-------------------!
```

FIGURE 2.17

end literal;.sp 2

An advantage to such a re-representation of the problem is that the solution only involves a change to the database: the creation of stereotypes (see section 2.3, above). The finding suggests that novices find it easier to mould their data structures to programs that are not quite right than to debug a program which fails to reflect their intentions in writing it. Indeed, databases are often more informative of the project students' intentions than their programs. Here's a last example of a program from one of the project students:

.sp 1;.literal

```
TO IMPLICATE /X/

1 PRINT "I'VE CAUGHT " /X/
2 NOTE /X/---IS--->GUILTY
3 PRINT "WHOM DOES " /X/ "WORK FOR?"
4 CHECK /X/---WORKSFOR--->?
  4A If Present: IMPLICATE *;EXIT
  4B If  Absent: PRINT "NOBODY, SO THE CASE IS CLOSED";EXIT
DONE
```

FIGURE 2.18

The structure is the familiar USE; GNO/S program that automatically makes everybody guilty of something. Here is the database that was designed to go with the program:

```
ANNIE...ISA...DRUGADDICT
 |
 |...BECOMESA...PROSTITUTE
 |
 |...WORKSFOR...TRAVIS
TRAVIS...ISA...PUSHER
 |
 |...WORKSFOR...INGRAM
INGRAM...ISA...SUPPLIER
 |
 |...WORKSFOR...JACKSON
JACKSON...ISA...MANUFACTURER
```

The database has what is a traditional structure for supporting programs of the type discussed in this Chapter; the overall structure of the project problem has been taken over wholesale. First of all there are the hierarchical relationships between the various actors in the devised drama:

```
X...ISA...<something-low-down-on-the-scale>
|
|...WORKSFOR....Y
Y...ISA...<middle-link-sort>
|
|...WORKSFOR....Z
Z...ISA...<something-high-up-in-the-scale>
```

The structure provides a simple framework within which the student can think about the problem in hand, but only up to a point. The program outputs 'ANNIE IS GUILTY', 'TRAVIS IS GUILTY', 'INGRAM IS GUILTY', and 'JACKSON IS GUILTY'. Of what can Annie possibly be guilty? Presumably the question occured also to the student who designed the program and the database. A drug addict works for a pusher the way a man with cancer works for a doctor. 'ANNIE BECOMESA PROSTITUTE' is inserted to justify the WORKSFOR relationship linking her to TRAVIS. Annie is a drug addict who becomes a prostitute - presumably to support her habit. The database itself does not clearly reflect what the student actually knows, but even less does the program reflect what the student 'meant'. Both in the design of databases and programs, the majority of project students show what Nelson, et. al. have called a "developmental time lag between comprehension and production" (Nelson, Rescorla & Gruendel, 1978).

CHAPTER 3

THE ORGANIZATION OF PROGRAMMING KNOWLEDGE

3.1 INTRODUCTION: In Chapter 1 the concept recall task of McKeithen et al. was described. They found that there was a correlation between particular organizations of concepts in memory and skill level: experts shared certain chunks; novices and intermediates shared a few chunks with experts, but not with one another. The argument made by McKeithen et al. was that there is something about the mental organization of concepts that produces performances of one type or another. This argument is plausible, and worthy of considerable testing. In this Chapter, two different experiments are reported. We want to know whether novices and experts differed in the organizations of their conceptual knowledge, and if so, how. We also want to know how conceptual organization and performance are related. The answer to the first question is that experts' organizations are as different from each other as they are from novices. The answer to the second question is that performance on these tasks is not a good predictor of novice programming performance.

3.2 A CONCEPT SORTING TASK.

In order to determine the organization of SOLO words, and of concepts more generally related to computing, a sorting task was designed. In order to validate the data generated, a second task, based on the Reitman-Reuter free recall paradigm, was performed by the Subjects taking part in the Sorting task. The concept sorting task and results will be described in the rest of this section. In the next section the concept recall task and results will be described. Following that, the relation between the sorting and the concept recall tasks will be described and the results from the two experiments will be compared.

The concept sorting technique required Subjects to build a nine-tiered hierarchical structure of concepts. The most general categories in terms of which a subject is capable of thinking about SOLO concepts are at the top of the hierarchy, and the more specific categories at the bottom. An example will make this clear. Imagine giving a person a list of objects and asking him to divide them into two piles and to think of category names for each. The person responds: Living-things and non-living-things. You ask him to now divide the list into three categories, and he responds: Plants and Animals and non-living-things. And so you go on until the person has created ten such categories, each having an identifying label. The minimum requirement for number of concepts in each pile on subsequent sorts was stringent enough to 'force' some reshuffling during the whole of the task. It was predicted that experts' trees would have an entirely different structure from the trees of novices. The prediction was based in the presumption that experts, having a good deal more experience with both SOLO specific concepts and the concepts related to computing in general, would have some ideal organization of concepts in memory, and that this organization would have an hierarchical structure. That is, it was expected that sorting the cards at successive levels would be a process of breaking up one group or another into two subgroups, and that there would be few 'cross-overs' resulting from the requirement to produce categories with at least some minimum number of cards in each. A cross-over is defined as the placing, at some sorting level, of a card into a pile which ends up having more than one ancestor in the unfolding graph. It was predicted that novices' graphs would have more crossovers, i.e., be less tree-like. The prediction was based in the presumption that novices' organizations would not be highly structured and that many 'cross-overs' would occur at the different levels of sorting.

METHOD: Subjects were given thirty three cards (4x6") with one concept name or SOLO keyword printed in the centre of each card, and instructed to sort the cards into two distinct categories and to provide a label for each of the categories. Each card had a different number (something from 1-33) written on the back for data collection purposes. The words printed on the 33 cards were these:

Proposition For-Each-Case-Of Database Inference Procedure Edit List Exit Bye Done Recursion If-Present Triple Node Variable Wild-Card Parameter Pattern-match Scope Association Iteration Next-Case If-Absent Network Symbol Relation Value Continue Check Describe Note Forget Print

When the sorting and labelling of two categories had been accomplished, the numbers on the back of the cards in each group and the labels provided for the groups were written down for later analysis. Then the Subjects were asked to create three distinct categories, again providing labels for each. This procedure was repeated again and again until each subject had created ten distinct categories. Each time the subjects sorted the cards, a minimum number of cards for each category was required. On the first sort, neither of the two categories could contain fewer than ten concepts. the piles. On the second sort, three categories, or piles, were required, again with a name for each pile, or category. This process was repeated, with one more category added at each subsequent sort, until the subject had created ten category piles. On different sorts, a different minimum number of concepts was required in any one pile. The minimum requirements for each sort were the following:

2 piles, 10 minimum in any one pile.
3 piles, 10 minimum in any one pile.
4 piles, 5 minimum in any one pile.
5 piles, 5 minimum in any one pile.
6 piles, 4 minimum in any one pile.
7 piles, 4 minimum in any one pile.
8 piles, 3 minimum in any one pile.
9 piles, 3 minimum in any one pile.
10 piles, 3 minimum in any one pile.

RESULTS and DISCUSSION: The novices' graphs are presented together in Figure 3.1. Experts' graphs are presented in Figure 3.2. Three individual raters were asked to sort the graphs. They were asked to look through them and to put those that looked alike into the same pile. Raters were told there was no restriction on the number of piles they

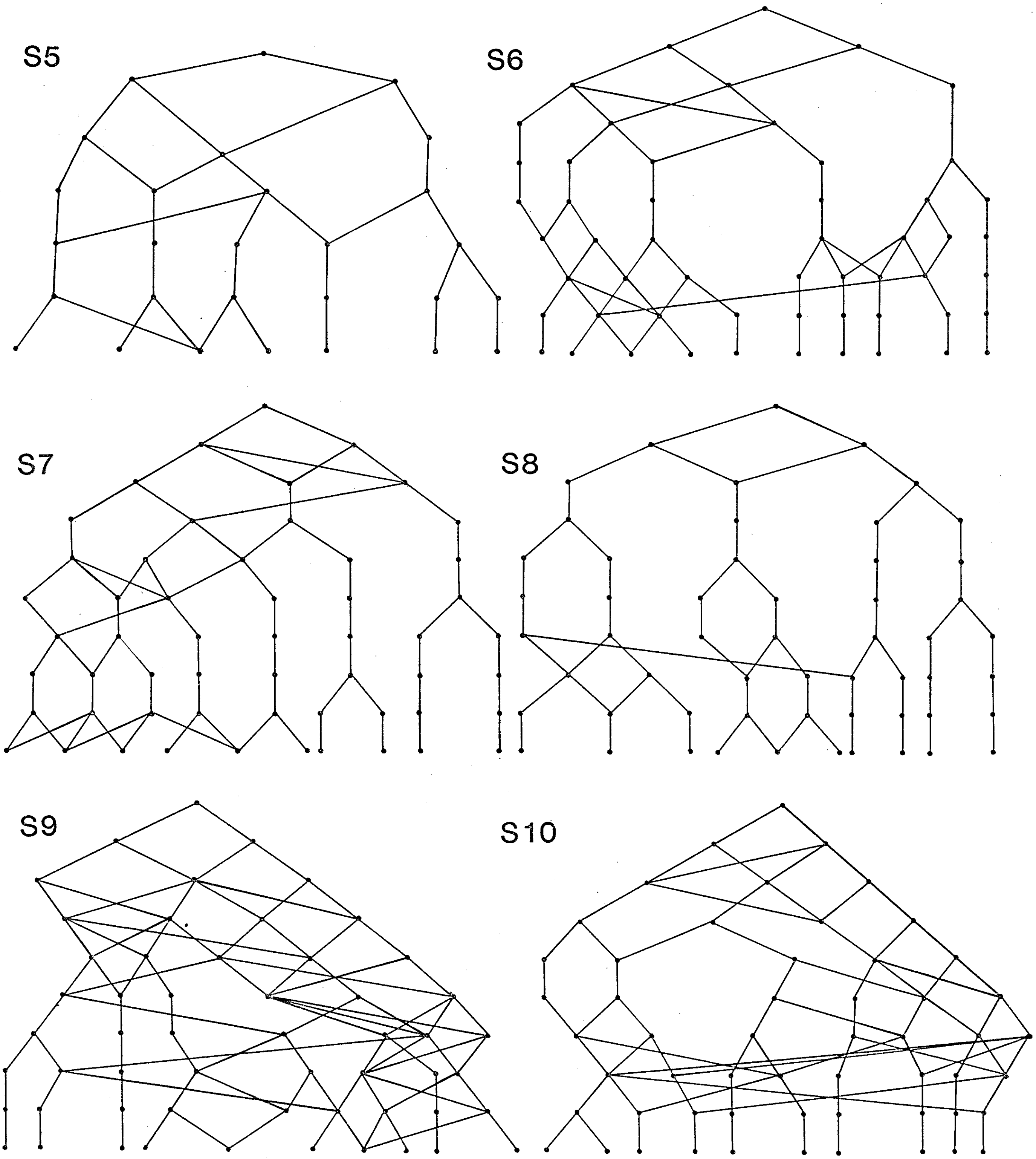
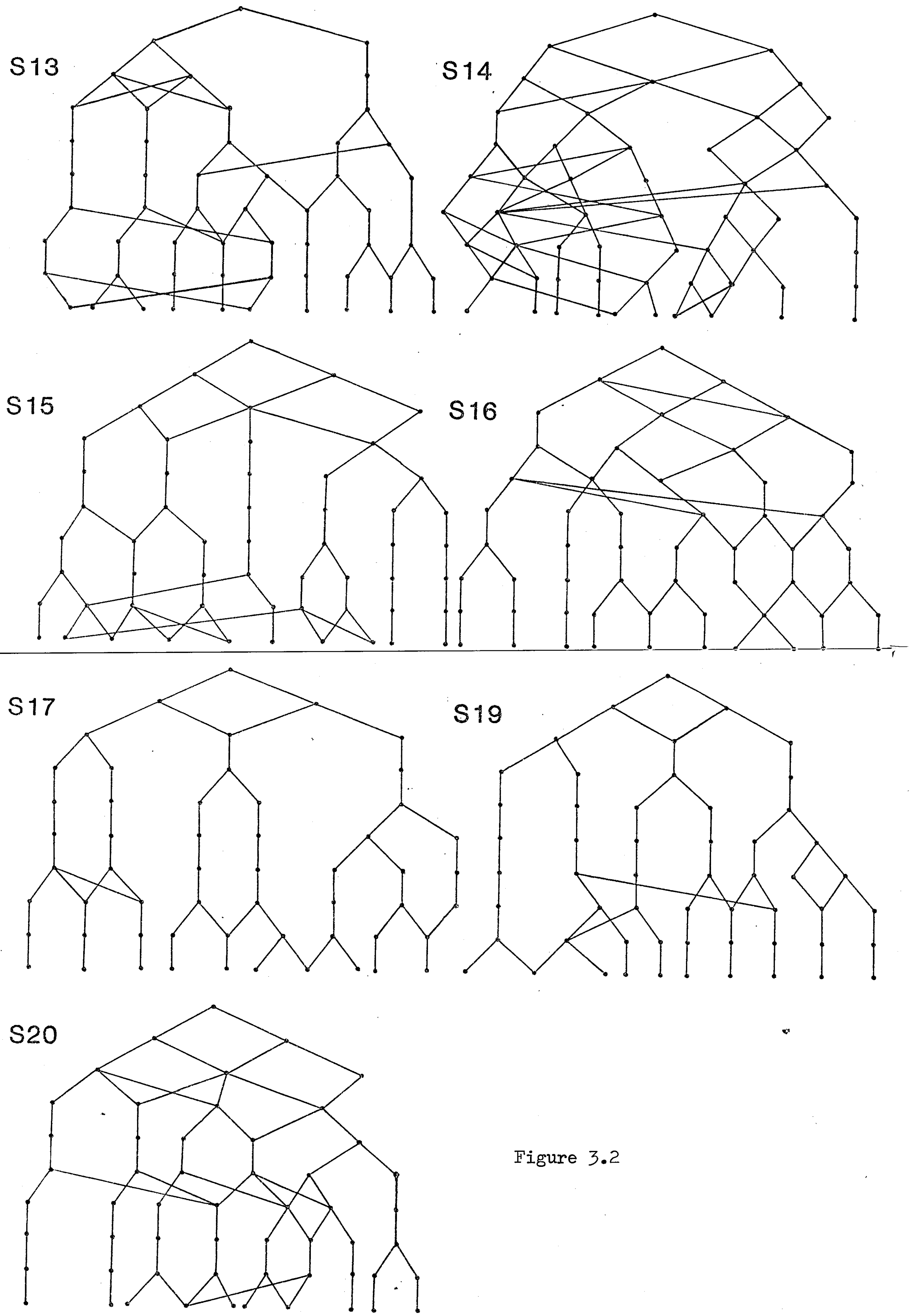


Figure 3.1



S13

S14

S15

S16

S17

S19

S20

Figure 3.2

could make. All three raters created four piles of graphs, and all three raters put the same graphs in the same piles. The rationale given by one of the raters (Henry H. Reuter, University of Michigan).

1) "A simple top half, a complex bottom:" [two novices (S5 and S8) and three experts (S15, S17, and S19)].

2) "A complex top, a simple bottom:" [one novice (S7) and one expert (S16)].

3) "A complex top and bottom:" [one novice (S6) and two experts (S13 and S20)].

4) "Just a mess:" [two novices (S9 and S10) and one expert (S14)].

There is clearly no way of distinguishing novices and experts in these reflections of the complexity of underlying organizations. There are some interesting features to the result, however. In Chapter 5, it will be seen that S8 outperformed all the other novices, as well as half of the experts, on a program transcription task. S5, on the other hand, will be seen to have been consistently one of the worst performers on the transcription task. In Chapter 8 the programming behaviour of two novices will be described in some detail. One of these, S8, will be seen to be quite a competent programmer on the given problem. The other (S5) will be seen to be quite deficient in programming knowledge. Yet, both were grouped together by all raters as producing similarly shaped graphs (global assessment). A closer inspection of the graphs reveals some interesting differences, however. S8's organizations are clearly more hierarchically structured. In fact, S8's tree is more like the graphs of Subjects S7, S15, S16, S17, and S20, in that they all contain polygons bounded by six lines, and like the graphs of Subjects S13, S15, S16, and S19 in that they all contain the 'tiered housing' structure so clearly seen in the right hand side of S8's tree. In contrast, S5's tree looks, by comparison with almost all the other graphs quite characterless.

The sorting data from one of the expert Subjects (S13, the designer of SOLO) was compared with that of all other Subjects by measuring the percentage of overlap within categories at three different levels (where 'level' is equal to the number of piles required by the sorting task): levels 3, 6, and 10. As the designer of SOLO, S13 was considered to be the 'ideal' expert on SOLO. The sorting data for Subjects S13 (the 'ideal' expert) and S8, for the level 6 data, are given below. The name for each category that was provided by the Subjects appears first, in upper-case letters, and the concepts that were placed in each category are contained in brackets below the category name.

S13:

SIX categories:

DATABASE FILE

(Prop. Database Triple Assoc. Network)

ITERATION FILE

(F-E-C-O Recur. P-Match Iter.)

RUBBISH

(Infer. Proc. N-Case Relation)

SOFT CORE PRIMS

(Edit List Exit Bye Done If-Pres. If-Abs. Continue)

NODE/VARIABLE

(Node Variable W-Card Param. Scope Symbol Value)

HARD CORE PRIMS

(Check Describe Note Forget Print)

S8:

SIX
categories:

SEMANTIC CORRELATES

(Prop. Infer. Assoc. Network)

FLOW OF CONTROL

(F-E-C-O Exit Done If-Pres. N-Case If-Abs. Continue Check)

DATABASE STRUCTURE

(Database Triple Node Relation Note Forget)

ABSTRACTpROCEDURAL

(Proc. Recur. P-Match Scope Iter.)

NON-STRUCTURAL COMMANDS

(Edit List Bye Describe Print)

ABSTRACTvALUES

(Variable W-Card Param. Symbol Value)

Overlap was computed by counting the cards which were sorted into the same group by both subjects, by the following rules. The first group produced by S13 was taken as the focus of attention. The first group was S13's 'DATABASE FILE':

DATABASE FILE

(Prop. Database Triple Assoc. Network)

The rater then tried to find the best fitting group in the sorting data at the same level for each of the other experimental Subjects in turn. For S8, the best fit was with that Subject's 'SEMANTIC CORRELATES' group:

SEMANTIC CORRELATES
(Prop. Infer. Assoc. Network)

where the concepts 'Proposition', 'Association' and 'Network' overlapped S13's 'DATABASE FILE' concepts. The amount of overlap, therefore, was 3 concepts (for S8) out of 5 (of S13's). Then a check was made to see if a better fit could be achieved by matching S13's 'DATABASE FILE' to other groups in (for this example) S8's data. If no greater overlap could be found, then the two groups (S13's 'DATABASE FILE' and S8's SEMANTIC CORRELATES' groups) would be eliminated from the data for purposes of comparisons between the Subjects' other groups.

The percentage overlap between S13 and all other Subjects is given in Table 3.1.

	LEVEL 3	LEVEL 6	LEVEL 10
(Novices)			
S5	79%	48%	61%
S6	67%	67%	55%
S7	48%	67%	64%
S8	67%	58%	48%
S9	64%	48%	48%
S10	73%	39%	61%
(Experts)			
S14	61%	61%	76%
S15	73%	73%	61%
S16	52%	64%	58%
S17	67%	64%	55%
S19	61%	52%	48%
S20	73%	61%	48%

TABLE 3.1

There were no significant differences between novices and experts in the amount of overlap of concepts with the 'ideal' expert at any level of organization. Interestingly, at both the most general level of conceptualization, and at the most specific (levels 3 and 10), S5 is more like the ideal expert in terms of amount of conceptual overlap (79% and 61%) than is S8 (67% and 48%). In terms of the argument put forth by McKeithen, et al. - that a corrollary of the acquisition of the experts organizations of knowledge is acquisition of the experts procedural skills - S5 should outperform S8 on various tasks such as program understanding, design, and so forth. As already indicated, this is not the case. At level 6, S8 has a greater overlap with the 'ideal' expert than does S5. Perhaps this is the correct level from which to make predictions about programming potential. Sadly, it isn't. Two other Subjects, S6 and S7, both have more overlap with the 'ideal' expert than S8, and yet S7 could not get anywhere in designing a program, and S6, who succeeded in imitating a program, could not explain the behaviour of the program she had designed. Still, as pointed out in the Introduction to this Chapter, and in Chapter 1, the notion that differences in knowledge organizations and skill differences go hand in hand is compellingly plausible.

As it turns out, there is an interesting difference between novices, and a relationship between experts and some novices in the sorting data: their ability to label the concept groups 'forced' out of them. The ability to provide labels that summarize the common core of a conceptual grouping is a high level skill. The comparison made here is between the novices S5 and S8, at the sixth level of the sorting data (any level could have been chosen and the outcome would have been the same).

S5's conceptual labels were these:

1) LOGICAL TERMS
2) INSTRUCTIONS USED AS PART OF ANOTHER PROCEDURE
3) DESIGN PART OF DATABASE
4) DOING PART OF DATABASE
5) INSTRUCTIONS THAT CAN BE USED ALONE TO CHANGE THE DATABASE
6) INSTRUCTIONS THAT CAN BE USED ALONE JUST TO DO SOMETHING

S8's conceptual labels were:

1) SEMANTIC CORRELATES
2) FLOW OF CONTROL
3) DATABASE STRUCTURE
4) ABSTRACT/PROCEDURAL
5) NON-STRUCTURAL COMMANDS
6) ABSTRACT/VALUES

The differences are clear. S8 quite apparently has a meta-language for thinking about SOLO words and more general computational concepts, and many of these would be familiar to experienced programmers - Flow of control, Database Structure, etc. S5, on the other hand, needs up to ten words to devise category labels, and even then it is not easy to see what the Subject has in mind when providing some of them - Instructions that can be used alone just to do something, Doing part of database, and so forth. In Chapter 7, the problem solving behaviour of both of these Subjects will be discussed in considerable detail. Of particular interest, with respect to the discussion here, is that during the problem understanding phase of problem solving (as described in Chapter 7) S8 begins problem solving with a 'method' (recursion) which was activated on contact with the first statement of the problem text, whereas S5 either remained bound to concepts actually mentioned in the problem text, or with lower level SOLO words.

The similarity between experts and novices demonstrated here really should not be seen as all that surprising. For instance, the differences reported by McKeithen, et al. (see Chapter 1) were largely differences between 'naive', as opposed to 'novice', and expert programmers. In fact, the only chunks shared by all the experts in the McKeithen study were 'THEN ELSE' and 'FOR STEP'. Nearly 9 out of 10 of their experts shared the chunks 'IF THEN ELSE', 'IF THEN', 'AND OR' and 'FOR STEP'. After this, the number of experts sharing chunks drops off dramatically. This seems not to be a very impressive distinction that can account for the differences between the programming skills of two different groups, especially when between 50% and 83% of all the novices also share these same chunks. The chunks IF-THEN, IF-THEN-ELSE, and THEN-ELSE, and the others mentioned here hardly constitute a sufficient basis on which to distinguish novices and experts in programming. Pigeons can learn to associate patterns of lighted discs, and novices can learn associations between programming concepts simply by noticing that certain of them tend to be discussed together in programming manuals. If a person stood around listening to mechanics in an automobile workshop for a while he would be able to tell you there was a closer association between 'gaskets' and 'cylinder heads' than between 'gaskets' and 'brake linings' simply because mechanics talk about head-gaskets and do not talk about brake-lining-gaskets.

The central argument of this thesis is that it is procedural and not declarative knowledge which distinguishes novice and expert programmers, or at least a combination of the two. The fact that a skillful novice such as S8 (and the evidence for the claim will be presented in various Chapters of the thesis) has less conceptual overlap with an 'ideal' expert than another novice who has almost no ability to solve programming problems suggests that the difference between expert and novice should not be measured as a distance between organizations of knowledge but in terms of the internal structure of the knowledge itself. The distinction has been pointed to as 'knowing that' and 'knowing how'. That is, two novices may associate recursion and iteration, for instance, but only one may know when and how to use the concept. This is the distinction we shall be looking into.

3.3 A CONCEPT RECALL TASK.

To determine whether the data from the sorting task, reported above, actually reflected durable, underlying organizations, rather than transient, or task dependent structures, another study was undertaken, replicating a design already in the literature.

The task is a multitrial free recall task (MFR) based on the Reitman-Reuter (1976) design, in which organization that occurs during learning is not a variable of interest. In the Reitman-Reuter MFR task data-collection begins only after the to-be-recalled list has been learned to perfection - i.e., after a robust cognitive structure has been constructed over the period of learning. In effect, Reitman Reuter have eliminated the bottom-up element from multi-trial list learning. Subjects are simply given the complete stimulus set and are invited to organize the elements before trying to commit them to memory. The technique is top down from the first recall trial. The method reveals the chunks of knowledge which individuals possess (the chunk data revealed by recall orders was verified by inter-response time data, and intuitively from knowledge of the domains of knowledge used in the different experiments carried out by Reitman Reuter, and by McKeithen). Here is how Reitman & Rueter explain their algorithm:

> "In free recall the subject traverses an ordered tree, beginning at the root node, and descends the nodes until a terminal item is reached. Upon recalling that item, he moves up to its immediate superior node and descends its constituent links until all of its descendent terminal items have been recalled. The order of processing of the contituents of any node must, of course, be consistent with its order type. Once all of that node's descendents have been recalled, traversal is resumed at the immediately superior node, and each of its constituents are descended until all of its terminal items have been output. Traversal continues in this fashion until all items of the required set have been recalled. There is only a limited number of recall orders that can be produced under these traversal constraints, and the terminal items subsumed by any node will appear as a chunk in each of them.
>
> Consider the example tree shown in Fig. 1 that represents a subject's presumed mental organization of items A-H. The recall rule prescribes that if the subject begins recall with item B, he will them recall C (completing chunk 5), then item A (completing chunk 2), then D-E-F and G-H. Or if the subject begins with G, he will then recall H (completing 4), then D-E-F (completing 3) then either A then B and C, or B and C then A (completing 5 before 2). There are exactly 16 linear orders of the objects that can be produced by such traversals, and they all preserve the essential chunk structure inherent in the tree.
>
> The technique is applied to data in the form of a set of complete recall orders. To collect appropriate data, subjects are asked to recall a large, well-learned set of items many times from many different starting points. It is important that the data include no omissions or intrusions, and that the subject give us a sample of the variety of orders he can produce. To induce variety, on some trails the subject is asked to recall freely (i.e., start with any item he choses) and on other trials he is required to start with a "cue" item and those that "go with it". This cueing breaks any recall stereotypy that may build up in a session, and encourages variety.
>
> From this set of cued and noncued recall strings, the algorithm efficiently finds the set of all chunks and

> represents this set as an ordered tree. In particular, the algorithm recursively examines the strings "top down" for chunks. The set of all such chunks forms a lattice which is then converted to a tree, with directionality indicated where appropriate. Equivalently, it can be written in the form of an expression with parentheses designating non-directional chunks, square brackets unidirectional chunks, and angle brackets bidirectional chunks.
>
> An important detail of this technique involves appropriate analysis of the cued trials. Since the cue items may be part of a chunk whose traversal is disrupted by the cueing process (directional chunks are particularly vulnerable), only that part of a cued trial that is assumed undisrupted should be analyzed. The disrupted and undisrupted segments of recall strings are identified in an initial step of the algorithm. First, the highest-level disjoint chunks - formed by the subtrees of the root of the tree induced by all recall orders, without regard for cueing - are identified. Second, in each string the effects of cueing are assumed to be limited to the highest-level chunk that contains the cue item. As a result, the part of each cued trial that involves traversal of the cued subtree is not used in the search for structure; only the latter parts, those involving natural traversal of the noncued subtrees, are used to build a second tree whose subtrees have the detailed structure induced from the noncued traversals. It follows that only noncued trials may be examined for the directionality of the root."

In the recall task the Subject organizes his recall by creating some optimum number of groupings of concepts, in order to make it easier to commit them to memory. The number of groups a Subject uses to organize the material for memorization suggests the level of conceptualization - from the more general to the more specific - the Subject 'feels at home with', i.e., again using the example from the previous paragraph, one Subject may prefer to organize his recall in terms of 'living-things' and 'non-living-things'; another may prefer to operate at the level 'wild-animals', 'domesticated-animals', 'pets', 'shrubs', 'trees', and 'flowers'. In the concept sorting task, the Subject is forced to reveal his organizations at all levels. Thus, level-six organizations 'fall out' of the organizations the Subject was forced to make at the five previous levels. The comparison of interest is the content of the chunks revealed under Reitman-Reuter paradigm and the equivalent chunking level from this experiment. For example, if the Reitman-Reuter algorithm indicates, say, 6 chunks for a particular subject, then we want to compare the contents of those six chunks with the sorting data at the level of six categories. In the Sorting task, the Subject will not arrive at his preferred level until he has sorted the concepts at five higher levels of conceptualization. If the contents of the chunks for both tasks are highly correlated, then there is on the one hand independent support for Reitman and Reuter's claim that their method reveals non-transient, underlying organizations, and on the other hand, there is support for the thesis that the higher levels of organization which are 'forced' from Subjects in the Sorting task are meaningful, since organization at the preferred level fall out of these organizations. The comparison of chunks is made by taking the number of chunks identified in the recall data and using the equivalent level of the Sorting data and testing the degree of association between the two.

METHOD: Subjects were asked to recall a set of 33 programming concepts and SOLO key words over 40 trials. Each word was printed on a 4x6 piece of card. The entire deck of 33 cards was shuffled and handed to the subject with instructions to learn the words so that he could perfectly recall all of them from memory. It was suggested that a good strategy would be to base learning around the creation of a number of categories into which the various words could be fitted. When the subject was able to recall all the words twice without error, starting wherever he liked, the experiment proper began. Over 33 of the next 38 trials the subject was given a different word to begin recall from, and asked to recall all the words that 'went with' the given word and then to recall all the rest of the words. On the remaining 5 trials the subject was asked to recall the words in any order he wished. If the subject made an error on any of the trials, that trial was repeated at the end of the 38 trials. In all, six experts and five novices took part in the experiment. Each Subject was tested separately. All recalls were recorded on audio-cassette tape for later transcription.

RESULTS: Ordered trees were derived for all Subjects, using the Reitman Rueter algorithm (the program was supplied by Rueter, University of Michigan). Then, a Chi square analysis was performed for each subject using both the Recall and the Sorting data. For purposes of illustration of the analysis technique, here are the chunks identified by the Reitman Rueter algorithm for S7 - 6 in all. The square brackets indicate uni-directional chunks; the round brackets indicate non-directional chunks; and the angle brackets indicate bi-directional chunks.

1: [DATABASE (TRIPLE NODE NETWORK) PROPOSITION ASSOCIATION RELATION]
2: [PROCEDURE NOTE (CHECK PRINT LIST DESCRIBE) EDIT FORGET]
3: [<IF-ABSENT IF-PRESENT> CONTINUE EXIT BYE DONE]
4: [RECURSION <ITERATION INFERENCE>]
5: [PARAMETER WILDCARD PATTERN-MATCH SYMBOL VALUE VARIABLE]
6: [FOR-EACH-CASE-OF NEXT-CASE SCOPE]

Since 6 chunks are indicated in the results from the Recall task, the equivalent 6th level of the Sorting data was used for comparison. This sixth level sorting data is presented immediately below the chunk data derived from the Recall data.

The organizations at the 6th level of the Sorting data were:

Level 6 = SIX CATEGORIES:
DATABASEPILE
Prop. Database Triple Node Network

```
INFERENCE PILE
 F-E-C-O Infer. Recur. If-Pres. Iter. If-Abs.
PROCEDURE-PILE
 Proc. Edit List Check Describe Note Forget Print
COMMAND-PILE
 Exit Bye Done Continue
PATTERN-MATCH-PILE
 Variable W-Card P-Match Symbol Value
RUBBISH-PILE
 Param. Scope Assoc. N-Case Relation
```

The degree of association between the chunks revealed by the two different tasks is very high (Chi squared = 25.557, $p < .001$; the Coefficient of Association is C = 00.66). The data from all Subjects were treated in the same manner, and for all analyses the significance was always better than $p < .025$. The analysis indicates that the Sorting technique discloses organizations similar to those disclosed usint the Reitman Rueter technique, plus organizational information at a number of other levels, both above and below that which is revealed by the Recall task. The advantage of using the Sorting task is that it is much less tedious or exasperating than the multi-trial free recall task.

Considerable overlap of conceptual organizations between the members of the different skill groups does not mean that novices and experts know the same things - in fact, the results suggest that even experts between them do not know the exact same things. The results do suggest that at a surface level, the level of representation of external relational structures, novices and experts are quite alike. When novices reach the point at which they can fairly successfully parrot more experienced programmers, different methods are necessary to discover what they know and what they yet have to learn.

DISCUSSION: Often, the tasks set up to determine what it is that distinguishes novice and expert programmers are not 'culture fair', favouring, as they do, the Knowledge of expert programmers. Such tasks can only teach us the obvious: experts Know more than novices. We should not need to be taught that lesson over and over again. What we need to Know is what the novice can do and what processes are brought to bear in the doing. The remaining Chapters in this thesis are attempts to bring novice programming behaviour within the field of vision, to try to understand them in their own terms. Sometimes information can be gained by comparing novices with experts, but other times the best source of information about novices is gained by looKing at their individual performances. Both of these approaches will be attempted in the following Chapters.

CHAPTER 4

MODELS OF RECURSION

4.1 INTRODUCTION

The task discussed in this chapter was designed 1) to test the hypothesis that novices and experts differ in terms of their respective models of recursion as a process, and, 2) to try to discriminate the models of recursion which novices actually do possess.

The conceptual model presented to students in the D303 Programming Manual defines recursion as a process that is capable of triggering new instantiations of itself, with control passing forward to successive instantiations and back from terminated ones. This is the model of the recursive process that experts are hypothesized to have. A graphic representation of the model provided in the Manual is depicted in Figure 4.1

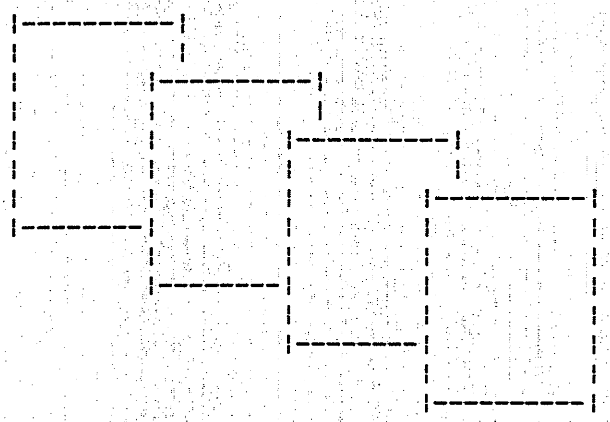


FIGURE 4.1. A graphic depiction of the 'Copies' model of recursion.

Students, on the other hand, are hypothesized to have a 'looping' model of recursion. That is, they view a recursive procedure as a single object instead of a series of new instantiations, having the following features:

1) an 'entry point', the constituents of which are the procedure's name and a parameter slot;

2) an 'action part', which is designed to add information to the database (by way of the 'NOTE <pattern>' in Figure 4.2, below);

3) a 'propagation-mechanism' for generating successive database nodes and feeding the values of these successive nodes back to the 'front part', or 'entry point' of the procedure. This 'looping' model is illustrated in Figure 4.2.

```
|-----------------------------------|
|                                   |
|                                   |
|   |--->PROCEDURE-NAME       /PARAMETER/           = 'ENTRY POINT'
|   |
|   |       NOTE  <node>  <relation>  <node>        = 'ACTION PART'
|   |
|   |     CHECK <node>  <relation>  <wild-card>   = 'PROPAGATION-
|   |      IF PRESENT:  PROCEDURE-NAME  *; EXIT      MECHANISM'
|   |                            |         |
|   |                          |         |
|   --------------------------           |
|                                        |
------------------------------------------
```

FIGURE 4.2. A graphic description of the 'Loop' model of recursion.

4.2 THE BEHAVIOUR OF PROGRAMS PREDICTED BY THE DIFFERENT MODELS

The hypothesis about differences between novice and expert models of recursion was tested by presenting Subjects with the Questionnaire in Figure 4.3, below. As may be seen, the Questionnaire contains three programs, called SOLUTION-1, SOLUTION-2, and SOLUTION-3, respectively. Two of these solutions are critical to determining a Subject's model of recursion, and in order to make clear to the reader the predictions being made I shall in this section demonstrate the reasoning processes presumed to occur when Subjects with different models of recursion are confronted with the critical two of the three programs presented in the problem statement, or Questionnaire. The text of the Questionnaire is:

PROPAGATING INFERENCES

Recently I needed a programme which would make the following inference: if somebody 'X' has 'flu, then whoever 'X' kisses also has 'flu, and whoever is infected spreads the infection to the person he or she kisses, and so on. Starting with the database given in Figure A, I needed a programme which would change the Figure A database into the Figure B database.

```
JOHN ISA MAN                            JOHN ISA MAN
 !                                       !
 !...KISSES MARY                         !...KISSES MARY
                                         !
                                         !...HAS FLU

MARY ISA WOMAN                          MARY ISA WOMAN
   !                                       !
 !...KISSES TIM                          !...KISSES TIM
                                         !
                                         !...HAS FLU

TIM ISA MAN                             TIM ISA MAN
 !                                       !
 !...KISSES JOAN                         !...KISSES JOAN
                                         !
                                         !...HAS FLU

JOAN ISA WOMAN                          JOAN ISA WOMAN
                                         !
                                         !...HAS FLU
```

Figure A   Figure B

I have been provided with three solutions to the problem, all called 'TO INFECT /X/' and these are labelled SOLUTION-1, SOLUTION-2, and SOLUTION-3, below. I want you to consider each programme in turn and say (A) whether or not the programme will do what I want it to do, and (B) if it will, say how it does it (in your own words), or, if it won't, say why it doesn't (again in your own words).

SOLUTION-1:

```
TO INFECT /X/
1 CHECK /X/ KISSES ?
 1A If Present: NOTE * HAS FLU ; EXIT
 1B If Absent: EXIT
DONE
```

SOLUTION-2:

```
TO INFECT /X/
1 NOTE /X/ HAS FLU
2 CHECK /X/ KISSES ?
 2A If Present: INFECT * ; EXIT
 2B If Absent:  EXIT
DONE
```

SOLUTION-3:

```
TO INFECT /X/
1 CHECK /X/ KISSES ?
 1A If Present: INFECT *; CONTINUE
 1B If Absent:  CONTINUE
2 NOTE /X/ HAS FLU
DONE
```

Please write your answers on the pages provided overleaf.
Thank you for cooperating.

FIGURE 4.3. The full text of the Questionnaire task.

SOLUTION-1 will not achieve the required effect. As may be seen, its outcome would be to add 'HAS FLU' to the node MARY after which the procedure would be terminated. SOLUTION-2 and SOLUTION-3 would both achieve the required output database. SOLUTION-2 works by side-effecting the database on the node first given as the argument to INFECT (= JOHN), and then generating the next node on the 'KISSES' list, which triggers the recursion. SOLUTION-3 works by creating a stack of bindings for /X/, ie. (JOHN MARY TIM JOAN) and side-effecting each on return from the recursive creation of the list (i.e., side-effects the listed nodes in reverse order).

4.2.1 The Effects of Running the Programs Under the Copies Model

Call the conceptual model presented in the Programming Manual the 'Copies' model of recursion, and the model hypothesized for students the

'Loop' model of recursion. Consider programs SOLUTION-2 and SOLUTION-3 from the perspective of either model.

A person possessing the Copies model should select SOLUTION-2 and SOLUTION-3 as programs that would achieve the intended results. Figures 4.4 and 4.5 provide a graphic display of the reasoning process which the Copies model was designed to inculcate, for SOLUTION-2 and SOLUTION-3 respectively.

Figure 4.4 shows the first two instantiations of the INFECT procedure. The parameter slot, at the first instantiation (marked (1) in the figure), has the value, 'JOHN'. At the first step of the procedure, the node 'JOHN' is side-effected by the addition of the description 'HAS FLU'. At the second step, the wild-card pattern match succeeds, binding 'MARY' to the wild-card variable, and at step 2A a new instantiation (marked (2) in the figure) of the INFECT procedure is triggered. The same process is repeated each time the pattern-match succeeds. Ascent from the recursion is indicated by the backwards pointing arrows in the figure, demonstrating when each instantiation is terminated.

```
  (1)
|-------------------------------|
|INFECT JOHN                    |
|1 NOTE JOHN HAS FLU            | (2)
|2 CHECK JOHN KISSES ? (= MARY) |
| 2A If Present: INFECT * ; EXIT| |-------------------------------|
| 2B If Absent: EXIT        ^   | |INFECT MARY                    |
|DONE                       |   | |1 NOTE MARY HAS FLU            |
|---------------------------|---| |2 CHECK MARY KISSES ? ( = TIM) |
                            |     | 2A If Present: INFECT * ; EXIT|
                            |     | 2B If Absent: EXIT        ^   |
                            |     |DONE                       |   |
                            |     |---------------------------|---|
                            |     |                           |
                            |-----|                           |----
```

FIGURE 4.4. The behaviour of the SOLUTION-2 program under the Copies model of recursion.

Figure 4.5 shows the last and penultimate instantiations of the INFECT procedure given in SOLUTION-3. The unwinding of the recursion is terminated (in (4)) when the wild-card pattern match (CHECK JOAN KISSES ?) fails. The 'If Absent' branch of the conditional contains the instruction to continue, and step 2 results in side-effecting the node 'JOAN' with the description 'HAS FLU'. Control is then returned to the flow of control statement at step 1A of the previous instantiation, (3), and the result is that the node 'TIM' is side-effected at step 2. The same process occurs for the two previous instantiations, of course.

```
      (3)
    |------------------------|
    |INFECT TIM              |                 (4)
    |1 CHECK ...             |            |----------------------|
    |1A If Present: CONTINUE |<---|       |INFECT JOAN           |
    |1B ...                  |    |       |1 CHECK ...           |
    |2 NOTE TIM HAS FLU      |    |       |1A If Present: ...    |
    |DONE                    |    |       |1B If Absent: CONTINUE|
    |-|----------------------|    |       |2 NOTE JOAN HAS FLU   |
      |                           |       |DONE                  |
 -----|                           |       |-|--------------------|
                                  |         |
                                  |---------|
```

FIGURE 4.5. The behaviour of the SOLUTION-3 program under the Copies model of recursion.

4.2.2 The Effects of Running the Programs Under the Loop Model

Figures 4.6 and 4.7 depict the operation of the Loop model in reasoning about SOLUTION-2 and SOLUTION-3, respectively.

In Figure 4.6(a) the procedure's parameter slot is first instantiated with 'JOHN'. At step 1, the side-effect to that node is accomplished, and at step 2 the wild-card pattern match succeeds, binding the value 'MARY' to the variable. As a result, at step 2A recursion is triggered, which, in terms of the Loop model, means looping back to the beginning of the procedure, taking along the value of the wild-card variable as the new value of the parameter slot. Since the parameter can contain only one value, the previous value is swept aside, as indicated in Figures 4.6(b) - 4.6(d).

FIGURE 4.6(a)

```
INFECT JOHN
1 NOTE JOHN HAS FLU
2 CHECK JOHN KISSES ? (= MARY)
2A If Present: INFECT * ; EXIT
```

FIGURE 4.6(b)

```
INFECT MARY
```

```
                                              |
                                              |---->JOHN
                                      1 NOTE MARY HAS FLU
                                      2 CHECK MARY KISSES ? (= TIM)
                                      2A If Present: INFECT * ; EXIT

FIGURE 4.6(c)                         INFECT TIM
                                              |
                                              |---->MARY
                                      1 NOTE TIM HAS FLU
                                      2 CHECK TIM KISSES ? (= JOAN)
                                        2A If Present: INFECT * ; EXIT

FIGURE 4.6(d)                         INFECT JOAN
                                              |
                                              |---->TIM
                                      1 NOTE JOAN HAS FLU
                                      2 CHECK JOAN KISSES ? (NIL)
                                      ---------------------------
                                      2B If Absent: EXIT
                                      DONE
```

FIGURE 4.6. The behaviour of the SOLUTION-2 program under the Loop model of recursion.

Strong evidence for possession of the Copies model would be selection of both SOLUTION-2 and SOLUTION-3 as correctly designed programs for the task in hand, plus some comment on the order in which the side-effect to the database occurs when SOLUTION-3 is run: since the side-effect occurs as the recursion unwinds, one would expect anyone who recognized this fact would mention it. Selection of SOLUTION-3 by itself would be weak evidence, at best, for possession of the Copies model, unless, again, some comment about the order of side-effects to the database is made. SOLUTION-2 should be easier for students to understand than SOLUTION-3, even if they have the Copies model, since SOLUTION-2 is in a form with which the students should be familiar: this program is an exact copy of the program used as an example of recursive procedures in the Programming Manual. Thus, it would seem improbable that anyone selecting SOLUTION-3 and rejecting SOLUTION-2 actually understood either program. In all cases, however, the Subjects' reasons for selecting or rejecting one and another of the programs were examined for direct evidence about the model possessed.

The behaviour of the SOLUTION-3 program, under the Loop model, is depicted in Figure 4.7. In Figure 4.7(a), at step 1 of the procedure, 'MARY' is generated by the wild-card pattern match, triggering recursion at step 1A. The result of this is that the value 'MARY' is 'fed back' to the beginning of the program. In 4.7(b), with the new value of the parameter slot = MARY, the value 'JOHN' is swept aside and the operation of the procedure produces 'TIM'. Figures 4.7(c) and 4.7(d) show the procedure looping through the chain of 'KISSES' relations until the wild-card pattern match (CHECK JOAN KISSES ?) fails. At that point (Figure 4.7(d)) the 'If Absent' branch of the conditional is taken, and the node 'JOAN' is side-effected with the addition of the description 'HAS FLU'.

```
FIGURE4.7(a)    INFECT JOHN
                1 CHECK JOHN KISSES ? (= MARY)
                 1A If Present: INFECT * ; CONTINUE

FIGURE4.7(b)    INFECT MARY
                         |
                           |---->JOHN
                1 CHECK MARY KISSES ? (= TIM)
                 1A If Present: INFECT * ; CONTINUE

FIGURE 4.7(c)   INFECT TIM
                           |
                         |---->MARY
                1 CHECK TIM KISSES ? (= JOAN)
                 1A If Present: INFECT * ; CONTINUE


FIGURE 4.7(d)            INFECT JOAN
                           |
                           |---->TIM
```

```
1 CHECK JOAN KISSES ? (NULL)
--------------------
 1B If Absent: CONTINUE
2 NOTE JOAN HAS FLU
DONE
```

FIGURE 4.7. The behaviour of the SOLUTION-3 program under the Loop model of recursion.

Strong evidence for possession of the Loop model would be:

1) selection of SOLUTION-2 as correctly designed, and,

2) rejection of SOLUTION-3, especially if this program is rejected on the grounds that only JOAN would be affected by running this program.

METHOD: The full intake of students (approximately 90) for the first week of the Cognitive Psychology Summer School (July, 1981) were given the Questionnairre in Figure 4.3 and asked to fill it in at their leisure and to return it to the experimenter. Students' previous experience of programming is not known. Nine experts also acted as Subjects in the experiment [Note 4.1]. The experts were Subjects S13 - S20, plus a research assistant, S21, who works in the psychology laboratory. S21 had a year of experience in writing Lisp and PASCAL programs.

RESULTS: Eight of the nine experts selected both SOLUTION-2 and SOLUTION-3, and one (S14) selected only SOLUTION-2 as the programs that would achieve the required output for the Questionnaire task. A typical expert's comments were those of S17:

> (Commenting on SOLUTION-2): "Yes. This procedure will transform the database to the required form from one call of INFECT JOHN. It does this by NOTEing that the node called as the procedure's

parameter, /X/, 'HAS FLU' as the first step. At the next step the database is CHECKed for a relation linking /X/ and another node by the relation KISSES. If such a 'triple' is found then INFECT is called recursively with the new node as the parameter /X/. This depth first search continues (like a real infection - although iteration must be involved in the real case, but not here) until the '/X/ KISSES ?' triple cannot be found and the EXIT route is taken (back through each level of recursion to the top). This solution is the most plausible representation of propagating inferences about infection given the database here."

(Commenting on SOLUTION-3): "Yes. This procedure will also transform the database as required using the single call INFECT JOHN. It does it in essentially the same recursive manner as SOLUTION-2 except that here '/X/ HAS FLU' is added to the database after the search for 'KISSES ?' has terminated. That is, the procedure recurses until no more '/X/ KISSES' ? triples can be found and as it EXITs back up through the recursions '/X/ HAS FLU' is added for each call of INFECT. This means of propagating is less psychologically plausible than SOLUTION-2 because of the high STS load imposed in this case."

Of the 90 or so students who were given the Questionnaire, 30 completed and returned it. In Figure 4.8 the graph represents the percentages of students (the stars on the graph) and experts (the plus signs on the graph) selecting particular programs as correctly designed solutions to the Questionnaire problem.

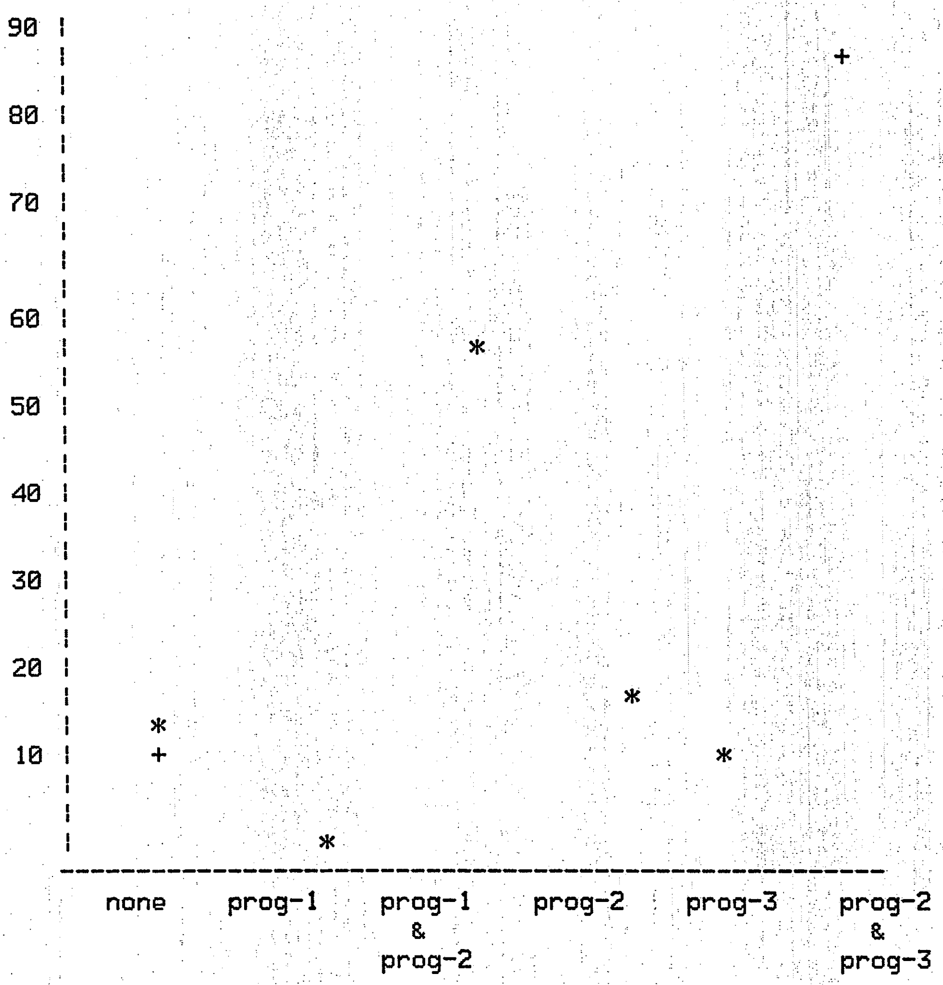


FIGURE 4.8 Percentage of novices and experts selecting different categories of 'correct programs'.

The actual number of Subjects represented by these percentages (student Subjects only) is provided in square brackets for each category of response, in Table 4.1.

1) None of the programs would behave as intended [4].
2) Only SOLUTION-1 would work [1].
3) Both SOLUTION-1 and SOLUTION-2 would achieve the intended output database [1].
4) Only SOLUTION-2 would work [16].
5) Only SOLUTION-3 would work [5].
6) Both SOLUTION-2 and SOLUTION-3 would achieve the intended output database [3].

TABLE 4.1

Table 4.2 shows the numbers of novices (N) and experts (E) who chose either SOLUTION-2, or both SOLUTION-2 and SOLUTION-3 as programs that would achieve the required effect. All other responses were collapsed into the category named OTHER. The difference in selection between novices and experts is highly significant (chi-squared = 21.40 p < .001).

Table 4.3 shows the number of novices and experts who selected only either SOLUTION-2, or SOLUTION-2 and SOLUTION-3 as correct solutions to the problem (the category OTHER has been removed). Novices chose SOLUTION-2 and SOLUTION-3 significantly less often than experts (chi-squared = 10.78, p < .01).

	N	E
SOLUTION-2	16	1
SOLUTION-2 & 3	3	8
OTHER	11	0

chi-squared with 2 degrees of freedom = 21.40
contingency coeff. = .59

TABLE 4.2

	N	E
SOLUTION-2	16	1
SOLUTION-2 & -3	3	8

chi-squared with 1 degree of freedom = 10.78
cont. coeff. = .52

TABLE 4.3

DISCUSSION: These results suggest that just over half the Subjects have adopted the Loop model of recursion (selected SOLUTION-2 and rejected SOLUTION-3), and that only three of the 30 Subjects have acquired the Copies model (selected both SOLUTION-2 and SOLUTION-3). Six of the Subjects appear to have understood little, if anything, about any of the programs (Subjects in categories 1, 2, and 3 in Table 4.1). There is also 'weak' evidence that a further one sixth of all Subjects have the Copies model (selected SOLUTION-3 as the only program that would do the task required). In order to determine more precisely how the different Subjects thought the programs behaved, an examination of their reasons for selecting and rejecting programs must be examined.

4.2.3 The Evidence For The Copies Model.

Of particular interest to us for the present is the small percentage (10%) of Subjects who thought both SOLUTION-2 and SOLUTION-3 were successful, indicating that just one in ten of the Subjects had the Copies model of recursion. And even this assumption may not be soundly based. The reasons given for selecting SOLUTION-2 and SOLUTION-3 by the three Subjects were the following.

S23, commenting on SOLUTION-2, thought:

> Yes, this programme will solve the problem, but will try to INFECT all but JOAN twice, so Step 2 is unnecessary.

On SOLUTION-3, S23 thought:

> Yes, this is the correct solution for this problem but could be achieved more easily by just using 'NOTE /X/ HAS FLU' because the 'KISSES' CHECK is not vital to the change, e.g 'JOAN HAS FLU' but does not KISS anyone. It's a 'Heads you win, tails I lose' flu situation!

Whatever S23 might mean by his comments on SOLUTION-2, I am not convinced he understands the program at all. Step 2 is, of course, necessary, as it carries the recursion. An explanation for this respondent's misconception that all nodes would be side-effected twice by the operation of SOLUTION-2, and that the "KISSES CHECK" in SOLUTION-3 is unnecessary, would be purely speculative and therefore no explanation will be attempted.

Here are S24's comments on both SOLUTION-2 and SOLUTION-3:

> Solution 2 and 3 look okay but I don't know if just typing in causes SOLO to change the database or whether the operator has to put in the new data - after which SOLO will INFECT new nodes who are kissed? P.S. Well I was told that 2 & 3 were okay! Before that I was wondering how JOAN got infected, but looking carefully I see TIM KISSES JOAN. As my favourite sport is jumping [to conclusions] I'm not very good at AI.

S24's response should fail to convince anyone that this Subject has any understanding of the behaviour of either program.

The last Respondent, S29, commenting on SOLUTION-2, and subsequently on SOLUTION-3, thought:

> (SOLUTION-2): Yes. Infects first argument, finds link, infects second argument and follows chain of infection.
>
> (On SOLUTION-3): Yes. Infects first argument and follows link as above. Only difference is order of carrying out instructions. (And, incidentally, of inferring.)

Only this Respondent, of the three, has a convincing story to tell, convincing especially in his determination of the order of the side-effects that occur as a result of running the procedure.

SUMMARY: Three Subjects out of thirty selected both SOLUTION-2 and SOLUTION-3 as correct programs, which was argued to be strong evidence for possession of a Copies model of recursion. However, the comments made by two of the three Subjects cast doubts on their level of understanding of either of these solutions. In the end the data suggest that only one in thirty Subjects after their initial training in SOLO programming, has acquired an expert's understanding of recursion, the Copies model. The 'weak evidence' for the Copies model will be discussed later in this chapter, in a subsection on 'Odd Models', because, rather than suggest that more Subjects possess the Copies model than indicated by the evidence examined in this section, the comments made by the Subjects who selected only SOLUTION-3 as correct suggest: 1) that many Subjects may have acquired idiosyncratic notions about recursive procedures, and 2) that some Subjects may have difficulty even recognizing recursive procedures.

4.2.4 The Loop Model

Most of the Subjects chose SOLUTION-2 as the only program that behaved as required, which is considered to be only weak evidence that the Loop model has been acquired by just over half (53%) of those who filled in the Questionnaire. Unfortunately, again, most of the comments made by these Subjects are less informative than necessary to work out their model in detail. A typical response was this:

> Follows 3 stages of:
> NOTE a) statement "X has flu"
> CHECK b) consequence Kisses flu
> INFECT c) spread of flu via Kiss.

Only three or four Subjects made comments longer than a couple of sentences, and only one of these (S13) made specific reference to looping as a mechanism of recursion. (This alone, of course, does not by itself mean that this Respondent actually has the Loop model.)

> This does solve the problem. When asked to infect John it adds the relation 'has flu' to John, checks who he kisses, loops back to infect that person, noting first that the person has flu and looking for any person that this new person kisses. It continues to loop the routine until all the nodes that attach the kisses relation have been infected. This will change the database in Fig. 1 to that in Fig. 2.

4.2.5 The Strong Evidence for the Loop Model

Of the sixteen Subjects who selected SOLUTION-2, four rejected SOLUTION-3 on the grounds that only JOAN would get 'flu. Since it has been argued that this is the strong evidence for possession of the Loop model, the figures indicate that just 13% of all Subjects have this model of recursion.

Here are some of the comments of those who thought only JOAN would be side-effected by the operation of SOLUTION-3. (Note that two of the Subjects name 'JOAN' while it is inferred that the other two mean 'JOAN' by their comments).

S7 commented:

> Won't work because of sequence - SOLO is shortsighted - follows one line at a time. Infect John & Step 1a proceeds to Check Mary for 'Kisses', finds it, checks Tim, then Joan. Only then, when "Joan Kisses ?" is not present, will step 1b activate step 2 & give flu to Joan, then stop. John, Mary & Tim remain fluless.

S11 commented:

> No one has flu to begin with. This will simply check who kisses who, flu is not passed on. It cycles through 1 and 1A, /X/ only gets flu if /X/ KISSES is absent.

S13 commented:

> This does not solve the problem. The program will loop its way through the database in a similar fashion to solution 2 but will only note that Joan has 'flu.

S14 commented:

> The procedure will search through the database repeatedly checking who /X/ kisses? However, the procedure will not use the database to note who has flu when kissed by /X/ because of the position in the procedure of NOTE /X/ has flu. Only the last value of /X/ will be infected.

SUMMARY: The Copies model is not a viable candidate for what our students know about recursion. Only three out of thirty Subjects on the Questionnaire showed evidence for the copies model, and two thirds of this evidence did not stand up to scrutiny. On the other hand, only four of the Subjects appear to have the Loop model, on the evidence of the strong indicants of that model. Thus, what the Subjects know about recursion can be accounted in terms of either the Copies or the Loop model in only five out of thirty cases. The comments of many of the Subjects who have been classified as providing only weak evidence for the Loop model suggest that they have acquired something other than either the Loop or the Copies models, and so the weak evidence will be discussed in a subsection, below, on 'Syntactic', or 'Magic' models of recursion.

4.3 OTHER MODELS OF RECURSION

A possibility not yet considered, of course, is that students (or some of them) have no model of recursion, or a model different from either the Copies or the Loop model. Some of the Subjects clearly do not understand recursion at all. The evidence for this will be discussed in the subsection below on the 'Null' model. Some of the evidence suggests that a few Subjects have slightly idiosyncratic Copies or Loop models, and this will be discussed in the subsection on 'Odd' models. Another possibility, for which there are fragments of evidence in the comments of some of the Subjects, is that there is a sort of 'Magic' model of recursion - that a procedure having a particular structure just is a recursive procedure, and it performs a certain sequence of operations, although the actual behaviour of the process is a complete mystery. This model will be discussed in the subsection on the 'Syntactic', or 'Magic' model.

4.3.1 The 'Null' Model

In this class would fall those Subjects who said that none of the programs would work (category 1 in Table 4.1), and the single Subject who claimed that only SOLUTION-1 would work (category 2). Typical comments made by Subjects falling into these categories were these:

> S16:Won't work. This is because as a definition of the procedure "infect", it can hardly use that very procedure as part of the initial definition. This would probably be refused by the computer.

S21 thought:

> No. The procedure is cyclic (i.e. it uses itself) and hence illegal.

One of the Subjects in category 3 (the weak evidence for the Copies model category) rejected SOLUTION-2 on the grounds:

> Will not work. Fans out from /X/ i.e. iteration

but selected SOLUTION-3 on the grounds:

> Will work. Goes in depth through the kissing, i.e. recursion.

The Subject is correct in the latter statement but not correct in the former. A possible explanation for these comments is that the Subject understood the behaviour of neither program, and perhaps thought he was being tested on his ability to discriminate recursive and iterative processes, and just made a guess. The Subject's comments on SOLUTION-1 offer no help; he merely comments: "Will not work".

4.3.2 The 'Odd' Model

It was stated above that the comments of many of the Subjects classified as providing weak evidence for either the Copies or Loop model suggested that they had various odd notions about recursion. In this subsection some of the comments of these Subjects will be given and interpreted. The interpretations suggest that these Subjects do have models of the behaviour of the procedures given in the Questionnaire, but because they have idiosyncratic ideas about some features of the programs they do not correctly predict the behaviour of the programs.

S27 (from category four in Table 4.1) sounds as though he has the Copies model when talking about SOLUTION-3:

> Yes. Programme will go through Infection process till last person kissing anybody has been found - then will note all four as having flu.

(Note that S27 states that the procedure recurses until "last person kissing anybody has been found" (= TIM, not JOAN). S27 then indicates that 'all four' will be Noted as having 'flu. What S27 means by the first comment is that SOLUTION-3 recurses as long as the pattern '/X/ KISSES ?' returns a value for the wild-card pattern-match.)

When discussing SOLUTION-2, S27 has this to say:

> No. The programme would add 2 persons who had flu to the database, but EXIT prevents further infection occurring.

An unimpeachable interpretation of S27's thinking about these programs is not possible, but this last comment provides us with a vital clue. S27 apparently believes that step 2A of SOLUTION-2:

```
    If Present: INFECT * ; EXIT
is read by SOLO as:
```

'INFECT * and EXIT immediately.

That is, the stopping rule for the recursion is not the absence of a pattern in the database but the flow of control statement 'EXIT'. S27's model of the behaviour of SOLUTION-2 is something like:

INFECT JOHN
INFECT MARY and EXIT.

If so, his model of the behaviour of SOLUTION-3 would be:

INFECT JOHN
INFECT MARY and CONTINUE [the INFECT process]
INFECT TIM
INFECT JOAN.

The comments indicate that S27 has an idiosyncratic model of the stopping rule for recursion, but exactly which model of recursion the Respondent possesses is impossible to determine. The fact that S27 points out that the side-effect to the database will occur last is some evidence for his possessing the Copies model.

Other Subjects also seem to have S27's particular problem about recursion's stopping rule. Here are a couple of their comments.

S1: (Commenting on SOLUTION-2):

It's not this solution, because it would only INFECT one person, then EXIT.

S25: (Commenting on SOLUTION-2):

Solution 2 will not do the job either. Although /X/ has flu, the EXIT in 1A & 1B stops the infection process.

In all these cases, the notion acquired is that the flow of control statement, rather than the absence of a particular pattern in the database, acts as the stopping rule for recursion. Recall that in Chapter 2 we found that a number of student programmers opted for a 'Continue Regardless' solution to the 'Guilt By Association' problem. In Chapter 2 it was suggested that students conceptualized the 'bug' in 'Termination problem' programs as 'unreachable code', and that some characterized the fault in terms of the control statement 'EXIT' while others characterized the fault in terms of the non-equivalence of patterns associated with different database nodes. The former group, it was suggested, thus changed the 'EXIT' instruction on the 'If Absent' branch of a conditional segment to be 'CONTINUE', the result of which would be a 'Continue Regardless' type of program. A different possibility is that the 'Continue Regardless' programs were designed that way in the first place. S25's comments, above, on the SOLUTION-2 program, relate termination of the program to the 'EXITs' on the two branches of the conditional statement. It is possible that some of our students think that both branches require 'CONTINUE' for the program to work properly.

Among the Subjects who have been classified as providing weak evidence for the Loop model there are a number of comments suggesting that some have a variant of that model. For example, although only four of the Subjects who rejected SOLUTION-3 named JOAN specifically, six other Subjects indicated that this program would succeed in giving only one person 'flu. Two Subjects thought 'JOHN' would get 'flu. For example, commenting on SOLUTION-3, S10 said:

> All we will get is John has flu added to the database. We will follow through procedure finding out only who kisses who and not using the fact to infect anyone so the procedure is not making any inference at all.

It may be that the Subjects who chose 'JOHN' as the only person to get 'flu have a 'matching bias' - that is, they match the '/X/' in 'INFECT /X/' to the '/X/' in 'NOTE /X/ HAS FLU' at the second step of the procedure. The Subject correctly determined that the procedure would work its way through to the end of the chain of 'KISSES' relations, but presumably thinks that successive values of the wild-card variable somehow 'get lost' and that the original value of the parameter is saved and operated upon by the action at step 2 of the procedure.

The comments of three other Subjects were too obscure on the point to determine which node they thought would be side-effected, but they apparently have some model of the behaviour of the program, as they all predict that only one (or, "at most only one") node will be affected by the operation of the procedure. Here, for example, is a typical comment which fails to specify which node is affected:

> (S30, commenting on SOLUTION-3): No, at most only one person gets flu for no apparent reason.

Finally, one Subject, S15, also thought only one node would be affected, but the claim was that it would be 'MARY'. Here are S15's comments on SOLUTION-3:

> Won't work. Only Mary gets flu. Only one node is affected.

This Respondent's comments may be based in the judgment that the design only allows one node to pass to the second step of the program, plus an priori model of the spread of 'flu - 'JOHN' Kisses 'MARY'; 'MARY' is the first person to get Kissed; Kissing can cause the spread of infection; if anyone does 'get' 'flu, it should be 'MARY'.

4.3.3 The 'Syntactic', or 'Magic', Model

The possibility to be considered next is that there is a 'syntactic' or 'magic' model of recursion which a number of our students may possess. A syntactic model would be based in the structure of the 'INFECT' program (SOLUTION-2 in the Questionnaire), which was used in the Programming Manual as an example of a recursive procedure. Any program having the same structure as the 'INFECT' program would be 'recognized' as a recursive procedure. It is a 'magic' model in that, although the student may know what the procedure does, he has no idea how the procedure achieves its effects. The Magic model has both are:

1) the name of the procedure

2) the description to be added to the value of the parameter

3) relation names.

These variable features are enclosed in quotation marks in the program frame presented in Figure 4.9.

```
'PROCEDURE-NAME' /X/
1 NOTE /X/  '<relation1>  <node>'
2 CHECK /X/ '<relation2>' ?
    2A If Present: 'PROCEDURE-NAME' * ; EXIT
  2B If Absent: EXIT
DONE
```

FIGURE 4.9. A program frame for the Magic model of recursion.

Hints that some students do base their judgments about the behaviour of programs in the syntactic structure of the program come from many of the comments made by different Subjects. For example, S12 thought:

> The procedure will search through the data base repeatedly checking who /X/ Kisses. However the procedure will not use the database to note who has flu when Kissed by /X/ because of the position in the procedure of NOTE /X/ has flu.

Note that this Subject does not mention that one of the nodes would be side-effected if the procedure were run but indicates that none of the nodes will be affected by the operation of the procedure.

S14's comments were these:

> Won't work. When at 2 /X/ has flu, it is after the event at 1, therefore cannot affect it.

S17 commented, on SOLUTION-3:

> This will not give correct printout because the triple /X/ has flu needs to be before the Check procedure.

These Subjects appear to be sensitive to the position of program segments, and, if they have no model of the actual behaviour of recursion, such indicants may be vitally important in their judgments about programs.

SUMMARY: Different novices may acquire a wide range of models of processes like recursion, or may achieve no understanding at all of such processes, or may simply be able to identify a program as a member of a class if it has expected features in expected configurations. The bad news is that as many as four out of five may fall into the latter two of these three categories if the responses of this self-selecting group (they chose to fill in and return the Questionnaire) accurately reflect the level of understanding of one major concept by the entire population of novices who have had limited experience of SOLO programming.

4.4 CONCLUSION

A range of abilities is demonstrated in these results. Novices can be distributed into different classes according to the internal structure of the individual concepts they have acquired. The data show that at least some novices - probably a quarter - can, after a fairly brief training period in SOLO programming, identify and mentally simulate the behaviour of recursive procedures. That is, they have mental models of the way recursion behaves. The notion that concretization of abstract concepts is an essential component of successful problem solving is suggested by the experimental results of Mayer (1981). Mayer was able to show that less able, novice BASIC programmers performed better on a variety of tasks if they were given a concrete system model before they began to learn about the system. Able novices, in Mayer's study, did not improve their performance as a result of being given such a model: they presumably had their own means of devising such models. Mayer also discusses a finding of Resnick & Ford (1980) showing that young children often invent their own concrete models of processes such as subtraction when they are first introduced to such concepts.

The results indicate that different knowledge bases need to be 'plugged in' to an adequate model of novice programming behaviour. The average of the novice with a Copies model of recursion and the novice with a Null model is not the novice with the Loop or the Odd model. On a given task some performances will be severely memory data-limited (Bobrow & Norman, 1975), such as those given by novices with either Null, Syntactic, or Odd models of recursion. There will be many tasks, on the other hand, on which some novices will perform very well. If a person has a mental model of a process, even if it is at variance with the conceptual model of the process, he will be able to make predictions about the behaviour of the process, although perhaps not all the behaviour, and perhaps inaccurately (Norman, 1982; Collins & Gentner, 1982). That is, students who are able to develop a Loop model of recursion will be able to design procedures in terms of the model and understand unfamiliar programs by mentally simulating their behaviour in terms of the model. More importantly, possession of a model provides a person with a basis for debugging the model when confronted with a counterexample (Jeffries, 1982). Thus, some novice performances may be indistinguishable from expert performance (with the possible exception of the comments made by novices and experts about the knowledge they are using: experts are notoriously concise, in contrast to the commenting performance of novices (Byrne, 1977; Simon & Simon, 1980). The terseness of the experts comments has been attributed to the expert's lack of self knowledge (Solloway; Byrne) and to the suggestion that expert's have 'compiled' processes (Simon & Simon). The two explanations may come down to the same thing in the end. A difference to be expected, then, is not in the overall structure of the performance between 'able' novices and experts, but the length of the protocol given, because of the copious comments of the novice, which is presumably possible because the processes of the novice have not yet become compiled.

In chapter 7 it will be shown how providing one or another of the models discussed in this Chapter as a Knowledge base for use by problem solving processes would lead to the prediction of radically different performances on the same problem task, and in Chapter 8 we shall see how well the models of performance match the actual performances of two different novice programmers.

CHAPTER 5

A TRANSCRIPTION TASK

5.1 INTRODUCTION

Chase & Simon have argued that the chess expert's ability to reconstruct a briefly viewed configuration of chess pieces from a game in progress is due to a correlation between the complex attack, defence, and other relations existing between pieces on the board and the way a Master's knowledge of chess positions is organized in long term memory. Less able players have not yet overlearned these complex relational patterns, or do not even yet know of them, and so their discovery would involve considerable reasoning rather than simple, automatic recognition. In the domain of computer programming, Anderson (1980) has made the plausible suggestion that expert programmers have memorized a wide range of 'ideal' programs which they are able to recall and use in an analog fashion when designing new programs. The store of ideal programs could also be used to recognize unfamiliar programs as members of a class known to the expert. It follows from this that novices, with far less experience than the expert, have been acquainted with and have memorized far fewer 'ideal' programs. Just as it has been suggested

(Simon & Simon, 1980) that a single piece on a chess board, rather than a group of pieces and their relations, constitutes one chunk for a novice at chess, it is also plausible to suggest that something less than an entire program is the primary unit of analysis for novice programmers when they are viewing a previously unseen program. In order to find out what aspects of an unfamiliar program are perceived by novice and expert SOLO programmers within the first few seconds of viewing, a transcription task was designed. It has been claimed (Newell & Simon, 1972) that what a chess Master sees in the first few seconds of viewing a game in progress - its structure - is qualitatively different from what is seen subsequently. Thus, it is suggested that the meaningful initial unit of analysis for experts would be the structure of the program. That is, the expert computer programmer, when confronted with a previously unseen program, would automatically attend to the overall structure of the program in the first few seconds of viewing and would therefore recall details from all the segments of a program. The particular program of interest for this experiment has the following syntactic structure:

```
TEST pattern
 IF YES: <action>  ; <flow of control statement>
   IF NO:  <action>  ; <flow of control statement>
TEST pattern
 IF YES: <action>  ; <flow of control statement>
   IF NO:  <action>  ; <flow of control statement>
TEST pattern
 IF YES: <action>  ; <flow of control statement>
   IF NO:  <action>  ; <flow of control statement>
```

If the expert does automatically attend to the global structure of the program, then it would be expected that he would note that there were three 'TEST' segments and should be sensitive also to the control structure of the program. If the expert notices that there are three

'TEST' segments, he will not have to attend to redundancies within the segment, such as the 'IF YES:' and 'IF NO:' prompts. (Part of an expert's knowledge is that the 'TEST' segment is a very stereotyped structure and that certain aspects of the structure need not be attended to at all. This claim could be experimentally tested by presenting some 'TEST' segments with the 'IF YES:' and 'IF NO:' statements reversed, and predicting that significantly fewer experts than novices would notice the transposition.) Thus, for the expert there is a lot of return for very little effort. If he noticed nothing more than that there were three 'TEST' segments then he would be able to 'recall' 21 words, where a 'word' is defined as either a word or symbol, such as a delimiter. The expected recall of an expert who had noted three 'TEST' segments would be:

```
TEST
 IF YES:
   IF NO:
TEST
 IF YES:
   IF NO:
TEST
 IF YES:
   IF NO:
```

Also, even if the expert did not notice the actual control structure of the program, since it is being claimed that his 'recall' really is a mixture of actual recall and 'construction' from prior knowledge, he would be able to infer the control structure from an internalized ideal program, and have a good chance of being correct even if he hadn't noticed any of the control statements on first viewing the program. A 'lucky guess' could add as many as 6 more words to his performance. The presumption, however, is that because the expert is sensitive to the overall structure of a program, he would notice something of the control structure, and that, because he will not process redundant information, he will also be able to process much of the variable information in the program.

The unit of analysis for the novice, however, would be something less than the program as a whole, perhaps as little as a single program segment. This is not to suggest that novices are blind, or have tunnel vision. Although it is predicted that the unit of analysis for the novice would be a single segment, he would be expected to notice that there was more to a program than the small part he found himself attending to on first viewing the program. Peripheral vision alone might be enough to indicate that a second 'TEST' segment followed the one being processed, and so the novice's first recall could be expected to take the form:

```
TEST pattern
 IF YES: <action>  ; <flow of control statement>
   IF NO:  <action>  ; <flow of control statement>
TEST
 IF YES:
   IF NO:
```

From this it looks as though the number of words the novice could be expected to recall after a first, brief view of the program would perhaps equal the number of words in the recall of an expert who only noticed that there were three 'TEST' segments and correctly guessed about the control structure. However, as has already been indicated, it is expected that the expert would notice a great deal more than just the global structure of a program on first viewing, and so the total number of words recalled by the expert should be significantly greater than that of the novices. Thus, the following three predictions were made:

1) The number of segments recalled by the novice after a first viewing of a target program will significantly differ from the number of segments recalled by the expert.

2) That experts would recall significantly more than novices on first viewing of an unfamiliar program, measured in overall number of words and symbols recalled.

3) That more experts would succeed in accurately transcribing the entire program than would novices.

5.2 THE TRANSCRIPTION TASK

METHOD: Twelve Subjects - six novices (S5, S6, S7, S8, S9, and S10) and six experts (S13, S14, S15, S16, S17, and S20) - were presented with a booklet containing a poem by e. e. cummings and six programs. The programs were written in an abstract version of SOLO. Three were regular programs, and the other three had the words and symbols of each line scrambled. The poem was included at the beginning of the booklet in order to provide practice trials in which the Subjects could familiarize themselves with the experimental procedure. The poem and the three regular programs (the first two of which were borrowed from Sime, Green, Guest, 1973) are reproduced in Figure 5.1, below.

in Just-
spring when the world is mud-
luscious the little
lame balloonman

whistles far and wee

and eddieandbill come

running from marbles and
piracies and it's
spring

when the world is puddle-wonderful

```
(1)         (TEST-1):       TEST time limited
                             IF YES: TEST-2; EXIT
                             IF NO: CONTINUE
                            TEST liable to space sickness
                             IF YES: astrobus; EXIT
                             IF NO: space-cycle; EXIT

            (TEST-2):       TEST conveying baggage
                             IF YES: spaceship; EXIT
                             IF NO: CONTINUE
                            TEST cost limited
                             IF YES: TEST-3; EXIT
                             IF NO: CONTINUE
                            TEST destination greater than 10 orbs
                             IF YES: superstar; EXIT
                             IF NO: cosmocar; EXIT

            (TEST-3):       TEST destination greater than 10 orbs
                             IF YES: hyperdrive; EXIT
                             IF NO: satellite; EXIT


(2)         (TEST-1):       TEST hard
                             IF YES: TEST-2; EXIT
                             IF NO: CONTINUE
                            TEST tall
                             IF YES: chop fry; EXIT
                             IF NO: CONTINUE
                            TEST juicy
                             IF YES: boil; EXIT
                             IF NO: EXIT

            (TEST-2):       TEST green
                             IF YES: peel roast; EXIT
                             IF NO: peel grill; EXIT


(3)         TEST flat has 2 bedrooms
             IF YES: looks promising; CONTINUE
             IF NO: too bad; EXIT
            TEST rent less than  100
             IF YES: really looking good; CONTINUE
             IF NO: too expensive; EXIT
            TEST neighbours are friendly
             IF YES: keep looking; EXIT
             IF NO: take it; EXIT
```

FIGURE 5.1. The practice poem and the three regular programs

used in the Transcription task.

Each subject was also supplied with five different coloured pens for transcription of the programs. On each trial, during an 'information extraction' phase, Subjects were allowed ten seconds in which to view one of the programs (or the warm-up poem). After ten seconds, Subjects would transcribe all that they could remember from the information extraction phase. Subjects were allowed an unlimited amount of time for transcription. When the transcription was completed for one of the trials, Subjects were allowed another viewing of the target program, for a total of five viewings. The Subject was not allowed to begin transcription until the entire ten seconds of viewing time had elapsed. They also were required to view the program over all the five trials. On each trial, Subjects used different coloured pens for transcription purposes. This procedure opens a window onto the different subjects' strategies for extracting information from unfamiliar programs over a series of viewings, since each trial was, as it were, colour coded. Finally, it was stressed that Subjects should put down everything they could remember or that they thought they remembered, or to guess at what they had seen. If they were wrong, they could always score a line through the error on subsequent trials and write in the correct words. Subjects were told that their task was to extract as much information as possible on each trial, and not to worry about errors that could be detected and changed on subsequent viewings. The point of this instruction was to encourage (especially novices) to reflect in their recalls what they 'knew' as well as what they had actually 'seen'. The full text of the instructions is given in Appendix B.

For purposes of analysis, only the results of the transcription of the third program were used. The decision to provide a fairly lengthy warm-up period was taken for the following reasons. First, in order to combat any effects of set that might be established during the first stage in which the Subjects were simply (unwittingly) being made familiar with the experimental procedure. Recall that, in order to familiarize the Subjects with the procedure a poem was presented for transcription. The standard way to read - and thus transcribe - a poem is to read first the first line, then subsequent lines. Because novices would be more likely to fall into such a set, given the nature of both the familiarization task and the novice's postulated unit of analysis, a lengthy warm-up period was thought to be essential in order to be fair to the novice Subjects. Second, a lengthy warm-up period gives Subjects opportunities to learn, and any initial differences between groups should be steadily reduced over the series of warm-up trials. If, after such a period, there are significant differences between the groups then the differences would seem not to be transient, or task dependent.

RESULTS: Table 5.1 shows the number of novices (N) and experts (E) who recalled details of all three segments of the program on the first trial. The experts were significantly drawn to extracting structural information about the program by comparison with the novices (chi-squared = 11.97, p < .001).

	N	E
ALL SEGMENTS	1	4
NOT ALL SEGMENTS	5	2

TABLE 5.1. The number of novices and experts recalling

information from the target program as a whole or in part.

Figure 5.2 shows a graph of the cumulative number of words recalled for both novices and experts over the five trials of the transcription task for the last program presented. Experts are indicated with an asterisk (e.g., *S13) on the graph.

```
60
59
58
57                   *S13         *S15/*S16      S6/S8           *S14/*S17
56                                               S10/*S17        *S20
55                                               *S20
54                                S6/S8/S9
53                                *S20
52                   *S15
51                                *S17                           S7
50                    S8
49
48
47                                               S7
46      *S13                                                     S5
45                   *S20/S9
44                   *S17         S10            *S14
43      *S15
42                                               S5
41
40                   *S16         S5/*S14
39
38
37      *S17         S6/
36                                S7
35
34                   S10
33       S8
32
31      *S20
30
29                   S5
28                   S7
27       S6/S9
26
25
24                   *S14
23
22
21
20      *S16
19
18
17
```

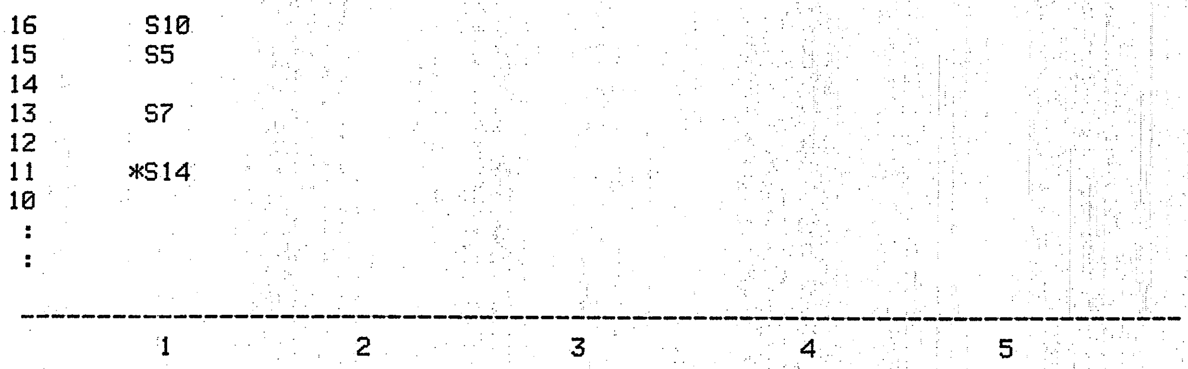


FIGURE 5.2. Total number of words recalled by both novices and experts for each of five trials in the transcription task.

Table 5.2 shows the same information in a different format. Table 5.2 also contains the rank order information for recall on the first trial. The differences between novices and experts, under a Mann Whitney test, are not significant (U' = 10, p .05). Nor are there statistically significant differences between the groups over subsequent trials.

NO. OF WORDS RECALLED IN TRANSCRIPTION TASK

NOVICES

TRIALS:	1ST	(RANK)	2ND	3RD	4TH	5TH
S5	15	(3)	14	11	2	4
S6	27	(6.5)	10	17	3	
S7	13	(2)	15	8	11	4
S8	33	(9)	17	4	3	
S9	27	(6.5)	18	9		
S10	16	(4)	18	10	12	
	----	----	----	----	----	---
TOTAL	131	(31)	92	59	31	8
MEAN	21.8		15.3	9.8	5.1	1.3

EXPERTS

TRIALS:	1ST	(RANK)	2ND	3RD	4TH	5TH
S13	46	(12)	11			
S14	11	(1)	13	16	4	13
S15	43	(11)	9	5		
S16	20	(5)	20	17		
S17	37	(10)	7	7	5	1

S20	31	(8)	14	8	2	1
	----	----	----	----	-----	---
TOTAL	188	(47)	74	53	11	15
MEAN	31.3		12.3	8.8	1.8	2.5

TABLE 5.2. Number of words recalled on all trials by all Subjects on the Transcription task.

Table 5.3 shows the number of novices (N) and experts (E) who succeeded in correctly transcribing the entire program. Significantly many more experts completed the task than novices (chi-squared = 11.97, p < .001).

	N	E
COMPLETED	2	5
NOT COMPLETED	4	1

TABLE 5.3. The number of novices and experts who successfully transcribed the entire target program.

Table 5.4, below, contains the actual recall of all twelve Subjects on the first trial.

```
*S14:   TEST flat has 2 bedrooms
         IF YES  looks promising
         IF NO                    CONTINUE

S7:     TEST flat has 2 bedrooms
         IF YES: looks promising
         IF NO:

S5:     TEST flat has 2 bedrooms
         IF YES: looks good: CONTINUE
         IF NO: too bad

S10:    TEST flat has 2 bedrooms
         IF YES: looks promising: CONTINUE
         IF NO: too bad: EXIT
        TEST

*S16:   TEST flat has 2 bedrooms
         IF YES: looks promising; CONTINUE
```

```
          IF NO: too bad; EXIT
         TEST

S9:      TEST flat has two bedrooms
          IF YES: looks promising; CONTINUE
          IF NO: too bad; EXIT
         TEST if less than  100
          IF YES
          IF NO

S6:      TEST flat has 2 bedrooms
          IF YES: this looks promising; CONTINUE
          IF NO:          bad; EXIT
         TEST rent under  100
          IF YES:
          IF NO:

*S17:    TEST flat has two bedrooms
          IF YES                    ; CONTINUE
          IF NO                     ; EXIT
         TEST
          IF YES                    ; CONTINUE
          IF NO                     ; EXIT
         TEST
          IF YES                    ; EXIT
          IF NO                     ; EXIT

*S20:    TEST flat has 2 bedrooms
          IF YES:
          IF NO:
         TEST rent less       100
          IF YES:
          IF NO:
         TEST neighbours are friendly
          IF YES:
          IF NO:

S8:      TEST: flat has two bedrooms
          IF YES: looks promising ; CONTINUE
          IF NO: too bad ; EXIT
         TEST:
          IF YES:
          IF NO:
         TEST:
          IF YES:
          IF NO:

*S15:    TEST flat has 2 bedrooms
          IF YES: looks promising; CONTINUE
          IF NO:                    ; EXIT
         TEST rent  100
          IF YES:                   ; CONTINUE
          IF NO:                    ; EXIT
         TEST neighbours friendly
          IF YES:                   ; EXIT
          IF NO:                    ; EXIT
```

```
*S13:   TEST flat has 2 bedrooms
         IF YES: looking good ; CONTINUE
         IF NO:               ; EXIT
        TEST rent less than  100
         IF YES: looks promising OK; CONTINUE
         IF NO: too expensive; EXIT
        TEST neighbours are friendly
         IF YES:
         IF NO: take it; EXIT
```

TABLE 5.4. The first trial transcription of a target program by novices and experts in the Transcription task.

DISCUSSION: Experts do attend to different aspects of an unfamiliar program in the first few seconds of viewing than novices. This is demonstrated by the finding that the experts, in contrast to the behaviour of the novices, generally recall some of the details of all segments of a program. The finding that there is no difference between novices and experts in total number of words recalled on the first trial seems to suggest that experts need to attend rather more closely to the overall structure of an unfamiliar program than hypothesized in the discussion in the Introduction to this Chapter. There, it was suggested that all the expert had to do was 'notice' the number of 'TEST' segments, and could then devote his attention to the control structure of the program and the variable content. Two of the experts (S15 and S17) did extract all the control information and another (S13) extracted all but one control statement. One possible explanation for the results on total number of words recalled on first viewing is that attending to the control structure takes up considerable time - most of ten seconds. A different possibility is that genuine differences between the groups, in terms of total number of words recalled on the first trial, have been disguised by the performances of one of the experts and one of the novices. The 'worst performance' by one of the experts is that of S14 who recalled the least number of words and recalled details from only one program segment. The 'best performance' by one of the novices is that of S8 who recalled the third highest number of words and was the only novice who recalled details, including control information, from all three segments of the target program. Such uncharacteristic performances are called 'outliers' and the dangers of including the data from grossly uncharacteristic performances has been commented elsewhere (Sheil, 1981). If the data from S8 and S14 are removed the difference between the groups is significant at $p < .025$ (Mann Whitney, $U = 2$). A glance at the recall information in Table 5.4 confirms that two of the six experts recalled considerable amounts of variable information plus all or almost all of the structural and control information in the target program. Another expert recalled most of the structural information and all of the control information. A fourth expert managed to extract all the structural information and some of the variable information in the program. Novices tended to extract most of the information from the first 'TEST' segment. Some were able to extract some of the structural and a small amount of the variable information from the second 'TEST' segment, but none of the novices, not even the exceptional case, managed to extract any of the control information beyond the first 'TEST' segment. The most surprising result is the performance of the two experts S14 and S16. S14 seemed to be unaware that there was anything more than a single program segment on the first trial. S16 was aware that there was more, but apparently was not able to use the information to his advantage.

5.3 IMPLICATIONS: At least on a task such as this, novices seem to be more comfortable working at the level of individual program segments. The analysis of programs in Chapter 2 provided a massive amount of data that was interpreted in such a way as to suggest that in designing programs the majority of novices also prefer or need to operate at the level of basic program segments. All together, the results presented here are consistent with the results from previous studies. Again differences between novices and experts have been demonstrated, and again a range of abilities within the novice group has emerged. One interesting finding is that the expert who was the poorest performer on the transcription task was also the only expert seemingly not having the Copies model of recursion. In later Chapters, it will be shown that the exceptional novice on the transcription task, S8, also excells in other endeavours by comparison with the performance of the other novices.

CHAPTER 6

REAL WORLD KNOWLEDGE VS. PURE PROGRAMMING KNOWLEDGE

6.1 INTRODUCTION: The material presented in this Chapter is meant to provide the groundwork for the 'interacting knowledge' model of problem solving that is to be presented in Chapter 7. In the first part of the Chapter a reading experiment is described, and a model of the understanding process is outlined. The story is in fact based on the 'BULLETHOLE' programming problem, which was mentioned in Chapter 1, and which is discussed in detail in Chapter 7. The 'BULLETHOLE' programming problem text has some statements which were designed to trigger programming knowledge, and some statements that were designed to trigger general knowledge of the world. In order to better understand the way in which programming and real world knowledge interact, the 'BULLETHOLE' problem statement was broken down into its different elements so that the role of each source of knowledge could be studied in isolation. In the first experiment, the 'BULLETHOLE' problem text was stripped of all reference to programming, and presented as a simple story to computer naive Subjects. In 4the second part of the Chapter, two short experiments are discussed. These experiments complement the story-stripped-of-programming experiment by looking at the effect of program-stripped-of-story. an abstract isomorph of the 'BULLETHOLE' problem was presented to novice programmers.

6.2 A MODEL OF STORY UNDERSTANDING.

In Chapter 2, the 'UNDERSTAND' model of the problem understanding phase was shown to be inappropriate as a model for understanding ill-defined, or open problems. In order to construct a model of understanding processes for ill-defined problems a number of Subjects were asked to read a story which described the following event: a person aims a very powerful handgun - a gun that will put a bullethole in anything in the path of the bullet - at a set of objects and pulls the trigger. The reading Subjects were asked to explain this story to the experimenter as they read it aloud, one line at a time.

Before entering into a discussion of the story understanding model I shall introduce and define a concept which has been used extensively (in one form or another) in recent models of story understanding, and which will be used throughout this and the following two Chapters of the thesis. The concept is 'schema'. A schema is defined as an abstract representation of a generic concept or event, such as a visit to a restaurant (Schank, 1975). The notion of a schema is also useful as a way of thinking about packets of problem solving processes (see below). Important characteristics of schemas, for the discussion of the protocol under consideration, are the following. First, schemas encode knowledge about stereotypic situations, actions, events, etc., rather than specific events (Rumelhart & Ortony, 1976). In story understanding, schemas are triggered by concept words mentioned in the story text. For instance, a story beginning "A man walked into the bank with a gun" would trigger schemas for a man walking, a bank, and a gun. There are different arguments about the processes involved in triggering schemas. Just & Carpenter (1980), and Thibadeau, Just & Carpenter (1982) have argued that schemas can be triggered by individual concept words at any point in the sentence comprehension process. Schank's group at Yale have used both this approach and the approach of first constructing a representation of the meaning of an entire sentence before script application mechanisms are applied (Schank & Reisbeck, 1981)

Another important feature of a schema is that it has slots through which it is linked to subschemata. The slots indicate the relational structure of the knowledge encoded in the schema. For example, a schema for 'PERSON' would have slots for 'SEX', 'APPEARANCE' and so on. The slots also have variables. Variables acquire default values through experience, but, whenever new instantiations of a schema are triggered, slots take on values given in the current event, whether it be a real world event or an event described in a story, either read or heard. For example, a schema for a man would have variable slots for important characteristics of a man, such as that he has two legs, arms, ears, and eyes; that he has a name, some particular height, and so on. The default value (in our culture) for height might be something like 5'10". If the schema is triggered by some event - say, a person reads "A car pulled up in front of Joe's cafe, and a man got out" - then the 'man' schema (amongst others) would be instantiated and, because there is no information to the contrary, the default value for the man's height would apply for the time being. If the story had contained the phrase, "a short man got out", then the default value would have been displaced, and replaced with the information that for this particular instantiation of the schema, the man was less than average height. The important point is that we can understand many things without having to be told all of them.

Still another important characteristic of schemas is that, once triggered, they activate procedures which either actively search for information or compute values for some of the variables in a schema, given the value of others (Palmer, 1975). A simple example, related to the story under consideration, is that our knowledge of guns includes information that guns are used to shoot things, and so, given the information that there is a man with a gun, then an expectation can be set up that the man will use the gun to shoot something or someone. Another important property of schemas is the constraint they bring to bear on inferencing processes (Schank, 1980).

Thus, in schema theoretic terms, understanding involves the triggering of schemas which function to constrain inferencing processes, to set up hypotheses (expectations) about future events, to provide default values, and so forth. There are other properties of schemas which are interesting and important, but for our purposes the above given sketch will suffice. (For interesting and more elaborate discussions of the schemas, and related notions, see Rumelhart, 1975; Rumelhart & Ortony, 1976; Friedman, 1980. For an example of the way the notions of variables and variable bindings can be shown to be powerful notions in story understanding when combined with various problem solving operations, see Collins, Brown & Larkin.)

Oftentimes, understanding involves more than instantiating a schema which accounts for an event, or sequence of events. Understanding sometimes involves inferring the goals a person is trying to achieve when he performs an action. If we see or read about someone running to a shop, we may infer that the person is trying to get to the shop before it closes, or is in a hurry to acquire something which is sold in the shop, and so on, because a method for getting someplace in a hurry is to run (and a method for getting to some place before it closes is to hurry). In other words, we see his actions as 'methods' for achieving goals. Various researchers in story understanding have proposed that an important understanding process is the application of such 'mean-ends' analyses to the events described in stories (Rumelhart & Ortony, 1976; Schank & Abelson, 1977; Collins, Brown & Larkin, 1980). Much of the behaviour of the Subject whose protocol will now be analyzed can also be understood in terms of this notion.

From one angle, then, the story is non-problematic. Everybody could be expected to understand what happened, the event described in the story. Models of such understanding processes have been constructed and simulated by computer programs (Schank, 1973, 1975).

From a different point of view, the story is hard to understand. Part of what the story 'means' is tied up with the purpose, or goal, of the person performing the action described (Schank & Abelson, 1977). Understanding from this point of view involves assigning a goal to the person performing the action. Therefore, the major part of problem solving for readers of the story is finding and assigning a goal to the story's protagonist. However, the story contains no useful context within which a model of the assigned goal could be easily evaluated. 1973). No reason for the protagonist's action is specified, either explicitly or implicitly, in the story. The story does not say why the person shot the pile of objects. Thus, it is predicted that it will be difficult for readers to understand, in the deeper sense, what the story is about (see Bransford & Johnson, 1972, 1973). The prediction is that very little concern will be shown by the reading Subjects over the layout of objects mentioned in the story, or in understanding the action that occured. Much more time would be expected to be devoted to attributing a motive for the action. In the next Chapter, as we shall see, these predictions would be expected to be reversed.

The model presented below is related to models of story understanding which have been presented elsewhere (Schank & Reisbeck, 1981; and especially, Collins, Brown, & Larkin, 1980). The difference between the model presented here and those presented elsewhere is that both domain related (programming) knowledge and domain independent knowledge are posited to interact in ways that will be described in detail.

METHOD: Five adult volunteers with different backgrounds (secretary, student, etc.) acted as Subjects in the experiment. The experiment was conducted in one of the experimental laboratories at the University. Subjects were seated in front of a tape recorder and the task was described to them in detail (see 'INSTRUCTIONS' in Figure 6.1). Subjects were asked to read each line of the story (see Figure 6.2) aloud, and to explain the story to the experimenter as they proceeded through each line. Once a Subject started reading the experimenter intervened only to prompt him to speak whenever there was a lengthy pause. The Subjects' comments were tape recorded and transcribed verbatim at a later date.

INSTRUCTIONS

> On the table in front of you there is a short story, which is covered with a plain sheet of paper. When we are ready to start, I'd like you to move the covering sheet of paper towards yourself, just enough to expose the first sentence of the story. I'd like you to read the sentence aloud, and then tell me what you understand about the story at that point. I want you to tell me everything you know about the story before you go on to expose another sentence of the story. I'd like you to tell me everything that occurs to you after you've read a sentence. You can look back to a sentence you've read previously whenever you'd like, but don't go forward to another sentence until you've explained everything you know, or think, up to that point. Is that okay? Do you understand? [Experimenter would at this point answer any questions that arose.]

FIGURE 6.1. Instructions to Subjects for the Story Understanding task.

1) Imagine a hypothetical world in which there are only the following objects: a sandwich, a plate, a newspaper, a book, a table and the floor.

2) The sandwich is lying in the centre of the plate, which is sitting on the newspaper, which is lying on the book, which is on the table, which of course rests on the floor.

3) Imagine someone standing beside the table with a .357 Magnum pistol.

4) The person puts the muzzle of the pistol on top of the sandwich, so the gun is thus pointing towards the floor.

5) A point three five magnum pistol will, when fired, put a hole through anything in the path of the bullet.

6) The person fires the pistol.

FIGURE 6.2. The text of the 'BULLETHOLE' story.

RESULTS: Two of the protocols were unsatisfactorily short. One Subject 'explained' each line by commenting that he understood the sentence. The second Subject commented that he couldn't take the story seriously. Therefore, both of these protocols were discarded from the analysis. The remaining three protocols were all quite lengthy, and one was chosen for detailed analysis. Although this is the only one of the protocols analyzed fully in this Chapter, it will be shown from a few extracts from the other two protocols that the model derived from the one protocol extends to the others.

Figure 6.3, below, contains the entire protocol for one of the reading Subjects during the reading phase of the experiment. As already mentioned, the Subject was asked to read each line aloud and to think-aloud about the contents of each line before going on to a subsequent line. The reading phase comprises all the comments the Subject makes before giving up on the task. This is the point at which the experimenter usually came in and probed for further information. The Subject's comments from this point onward are not used in the analysis, because of the dangers that any information derived under the experimenter's probing is subject to a number of different interpretations (see Ericsson & Simon, 1980; but also see Collins, Brown & Larkin, 1980). Comments on the contents of Figure 6.3 are provided immediately following its presentation.

An Example Protocol: SUBJECT 2.

1) 'Imagine a hypothetical state of the world in which there are only the following objects: a sandwich, a plate, a newspaper, a book, a table and the floor.'

The floor.	RE-READING
Can that be correct?	PROBE
If it's got a floor then surely it should have walls and a ceiling too.	CONFLICT
E: Is that what you think of it, nothing else?	E-COMMENT
Um....	?
Sandwich and a plate suggest that somebody is going to have something to eat.	MODEL-1
Not necessarily straight away, not necessarily that day, it could have been a long while ago.	ELABORATION
Obviously [was] "a sandwich, a plate, a newspaper, a book, a table and the floor".	RE-READING
Um....	?
Suggesting somebody is going to read.	MODEL-2
But you don't usually sit at a table, unless you sit on something, and there's no chair.	CONFLICT
Newspaper could suggest something as far as, um, a down and out, living there.	MODEL-3
Making do without a chair.	SUPPORT
But then, perhaps, hardly likely to have a sandwich.	CONFLICT
Um....	?

2) 'The sandwich is lying in the centre of the plate, which is sitting on the newspaper, which is lying on the book, which is on the table, which of course rests on the floor.'

Um....	?
Somebody's stacking them.	ELABORATION
Um....	?
I suppose it could be for a photographic session or for a still life.	MODEL-4/ MODEL-5
Um....	?

3) 'Imagine someone standing beside the table with a .357 Magnum pistol.'

This, um....	?
Brings into mind some sort of hide out, or....	MODEL-6
I don't know.	META-COMMENT
Um....	?
People don't usually stand beside tables, with pistols unless they have some intention of either using them....	PROBE
Uh, "standing beside the table with a pistol".	RE-READING

Could be some kind of shooting range.	MODEL-7
But hardly with a sandwich on it.	CONFLICT
Um....	?

4) 'The person puts the muzzle of the pistol on top of the sandwich, so the gun is thus pointing towards the floor.'

Um....	?
Why would he want to shoot a sandwich?	PROBE
It's hardly likely to jump up and bite him.	PROBE-FAILURE
Doesn't really tell me much.	EVALUATION
The chap might have a jolly good reason for wanting to shoot a sandwich but I really can't, I couldn't judge, not from that.	EVALUATION

5) 'A point three five magnum pistol will, when fired, put a hole through anything in the path of the bullet.'

The sandwich is going to silence it.	MODEL-8
Why would he want to shoot the rest of the things, the book, the table?	CONFLICT
Ah, um....	?
I can't understand why he would want to shoot through the plate.	CONFLICT
But if he wanted to shoot through the sandwich as a sort of a home made silencer, I suppose, um....	PROBE
I don't know what the significance of the plate would be.	CONFLICT
But the book would catch the bullet.	MODEL-9
Perhaps it's some kind of ballistics test, to see how the pin hits the end of the bullet.	ELABORATION
Uh, "put a hole in anything in the path of the bullet".	RE-READING
Finding out how strong it is.	MODEL-10

6) 'The person fires the pistol.'

Um....	?
Now I don't know why a person would want to fire a pistol, a shot, at "the sandwich, which is sitting on the plate, which is on the newspaper, which is on the book, which is on the table".	EVALUATION
Um, it doesn't tell me why.	EVALUATION
It tells me that these things happen, or has happened or will happen, but it doesn't tell me why.	EVALUATION

E: Is that it?	E-COMMENT

That's about all I can get from it, I'm afraid.	RESPONSE (E-COMMENT)

Um....	?
No....	?
It tells me it's certainly not a rifle range because he wouldn't be pointing it at the sandwich.	EVALUATION

He would be pointing it at....	ELABORATION
Um....	?
Which I thought it might have been if, um....	EVALUATION
It said "imagine someone standing beside the table with a .357 magnum pistol."	RE-READING
I thought it may have been some sort of rifle range.	HYPOTHESIS
Some what out moded but um....	EVALUATION
And the rest of it, it doesn't really illuminate it, it just tells me there's somebody there and they intend to, and do, fire at the sandwich.	EVALUATION
E: Is that all?	E-COMMENT
Yes, I can't see anything else unless he doesn't like the book.	MODEL-11

FIGURE 6.3. A think aloud protocol from Subject S2, while reading the 'BULLETHOLE' story.

In Figure 6.3, the protocol has been divided into six 'episodes'. Each episode is labelled (1-6), and contains, first, the line which the Subject read from the story, and, second, all of the comments the Subject made before going on to a subsequent line of the story. The Subject's comments are presented as sentence segments, where possible. Otherwise the segments are part sentences which trailed off into nothing, or the frequent, deliberative "Um"s which the Subject enunciated. Each segment of the protocol also has a label attached, the label indicating the kind of process the investigator thought the Subject's comment reflected. Altogether, there are 13 labels, or categories of response. The labels that were used to classify the protocol segments, plus their frequency of occurence are given in Table 6.1.

?	15
MODEL	11
MODEL-EVALUATION	9

MODEL-CONFLICT	7
RE-READING	5
PROBE	4
EXPERIMENTER COMMENT = (E-COMMENT)	3
ELABORATION	4
META-COMMENT	2
PROBE-FAILURE	1
MODEL-SUPPORT	1
RESPONSE TO QUESTION	1

TABLE 6.1. Classificatory Labels applied to Subject S2's reading protocol.

The major category of response to the various sentences in the story is that represented by the question-mark symbol. The question-mark indicates that the investigator was not able to tell what the Subject was thinking about. However, many of the unknowns are associated with the Subject's statement: "Um". Many of these occured either immediately following the reading of one of the lines from the story, or soon after a particular model (see below) of the story had been proposed. The "Um"s may reflect 'waiting time' while a model is being constructed, or 'evaluation' and other important information processes which are thus hidden from the investigator.

The second most frequently applied label is 'MODEL'. The 'MODEL' label was applied whenever it appeared that the Subject's statement offered a possible model of the events thus far encountered in the story. In the first episode, for example, (Model-1) the Subject comments that the mention of a sandwich and a plate "suggest that somebody is going to have something to eat".

Models are presumed to be based on schematic knowledge; some element(s) of a problem text triggers some schema (perhaps set of schemas) which serve as a basis for constructing a model of the story. The process involved in the construction of the model seems to be simply the triggering of the 'EAT' schema, which contains specifications that the AGENT slot be filled by a human (the expected AGENT for eating a sandwich). With very little evidence to go on from the information in the problem statement, the default value of the AGENT slot is instantiated ("somebody" is going to have something to eat). The model can be represented informally as the instantiation of default values in a few of the slots of an "EAT" schema, where the dollar symbol merely indicates that a schema is denoted:

```
$EAT
ACTION: an eat
GOAL: satisfy hunger
AGENT: a person
OBJECT: a sandwich
```

An 'EAT' schema would be considerably more elaborate than suggested here. The point, though, is that given so little information the reader seems unwilling to over commit himself to the 'eating' model and so develops a model by instantiating the fewest possible number of slots that would account for what is actually known at present. Note that in so doing, an actor is introduced into the story by the reader before one has been introduced by the writer.

The particular schemas in which models are based, and the text elements which triggered them, are often mentioned by the Subject. In the example above, sandwich and plate were indicated as cues for Model-2, based in the 'EAT' schema. In order to show which cues activate certain schemas, the two will be represented as productions, where a production is defined, for the moment, as a formalism for linking two sets of words or symbols by pointing from one to the other. Productions have a left hand side and a right hand side. Rules for construction are these: the left hand side of a production contains words from a story text. The right hand side of a production contains the name of a model of the story. The arrow linking the left and right hand sides is meant to indicate that the words in the left hand side of the production triggers, or activates, a schema responsible for the model named on the right hand side. In the example:

sandwich & plate => EAT

the left hand side of the production, containing 'sandwich & plate', is linked to the 'EAT' on the right hand side of the production. With this definition in mind, we can try to identify the schemas that are triggered as the Subject reads through the story text, and also to identify the words that trigger the schemas.

MODEL-1	sandwich & plate	=>	EAT
MODEL-2	newspaper & book	=>	READ
MODEL-3	? (MODEL-3) newspaper	=>	DOWN AND OUT PERSON
MODEL-4	on...on...on...	=>	PHOTOGRAPHIC SESSION
MODEL-5	on...on...on...	=>	STILL LIFE
MODEL-6	? & person & pistol	=>	HIDE OUT
MODEL-7	table & pistol	=>	SHOOTING RANGE
MODEL-8	sandwich & pistol	=>	SILENCER
MODEL-9	book & pistol	=>	BALLISTICS TEST
MODEL-10	put a hole in anything in the path of the bullet	=>	POWER TEST
MODEL-11	book & pistol	=>	DESTROY BOOK

Note that at least one of the triggers - that for MODEL-6 -

contains question mark symbols. This is merely to indicate that there are surely second order cues for schemas, and these second order cues have not been identified. That is, intermediating schemas are triggered by cues in the story and in turn trigger the schema reflected in the model (see, for example, Norman & Bobrow (1977).

The third most frequently occuring label is 'MODEL-EVALUTION'. In model-evaluation the Subject applies a current model of the problem to the text of the story for evidence that the model is correct. After reading the fifth line of the story the Subject can be seen to hold the strong expectation that there is some reason for the action of the story's protagonist, but the Subject is unable to discover anything in the details of the story which would permit evaluation of different proposals about what the motive might be. This is as predicted in the introduction to this section of the Chapter. As there is nothing in the story text but the description of a situation and an action performed, there is no support or disconfirming evidence about the goal of the person performing the action. Subsequent attempts at evaluation occur after the Subject has read the final line of the story. Model evaluation inheres in the categories 'Model-conflict' and 'Model-support', both discussed below.

Next comes 'MODEL-CONFLICT'. A conflict is the production of information which contradicts a model. For instance, the Subject develops a 'shooting range' model (MODEL-7) as a possible account of the mention of 'a person' (standing beside) 'a table', (with) 'a pistol'. When a model is proposed, certain hypotheses about its suitability are made available, and these hypotheses tested. For example, when the 'shooting range' model is proposed, the Subject tries to instantiate slots of a 'shooting range' schema with data from the story: the 'sandwich' is proposed as the value of the schema's TARGET slot, but the value is presumably wildly at variance with that slot's expected types of values. Model conflict occurs after Models 2, 3, 7, and 8 have been proposed.

'RE-READING' occurs at 5 different points in the protocol. Re-reading simply indicates that the Subject read a line from the text a second time. In all but one case (the last instance, episode 6) the Subject re-read the same line that had just been read.

The 'PROBE' label indicates that the Subject seems to be searching a fragment of knowledge for an answer to a specific question set up at some point in the ongoing processing. For example, after reading the first line of the story, the Subject first re-read the last couple of words of the sentence (The floor.). The words trigger a room schema which has slots for walls, floors, and windows, etc. The information in the first line of the story specifically contradicts (= Model-conflict) the knowledge the Subject has about the usual construction of rooms.

'ELABORATION', as the name suggests, involves using knowledge that has been accessed to elaborate on information in the story text. For example, after reading about the layout of the objects in the second line of the story, the Subject 'elaborates' the information by introducing 'somebody' who is 'stacking them'. Again, when the 'BALLISTICS TEST' MODEL is proposed, the Subject enlarges the story somewhat by proposing a reason for the 'ballistics test'.

There are only a few occurences of the various other labels; usually only one. 'PROBE-FAILURE', as the label suggests, reflects the failure to find sought for information during a probe of a knowledge structure. 'MODEL-SUPPORT' occurs when something in the story adds support to a model. In the single case of 'META-COMMENT', the Subject is commenting on his own state of knowledge ("I don't know."). The two remaining categories are 'EXPERIMENTER-COMMENT' and 'RESPONSE (E-COMMENT)'. The meaning of the labels is self evident.

THE ROLE OF PROBLEM SOLVING IN STORY UNDERSTANDING.

The pattern of responses to the different lines of the problem story in the Protocol in Figure 6.3 is not entirely uniform, but there is a lot of regularity. Elements of each line of the problem statement trigger some schematic knowledge which serves as a basis for the construction of a model of the events mentioned in the story. The model that is constructed is then evaluated, and, if the model is contradicted, rejected. Subsequent models are based either in new schemas triggered by story elements, or in a combination of elements of rejected, earlier models and newly triggered schematic knowledge. The processes involved in the construction of a model are binding, and re-binding, of schema slots to values, and evaluation of the bindings. The 'GOAL' slot of the model is repeatedly re-bound. Examples of such rebindings are these:

A) ACTION: eat
AGENT: a person
OBJECT: sandwich
GOAL: satisfy hunger

B) ACTION: read
AGENT: a person

```
        OBJECT: newspaper, book
        GOAL: inform self

C)      ACTION: hide
        AGENT: a person
        OBJECT: self
        GOAL: protect self

D)      ACTION: shoot
        AGENT: a man
        OBJECT: sandwich
        INSTRUMENT: a gun
        GOAL: silence noise of gun

E)      ACTION: shoot
        AGENT: a man
        OBJECT: book
        INSTRUMENT: a gun
        GOAL: determine characteristics of firing pin
        SUBGOAL: object 'catches' bullet, enabling agent to achieve goal
```

In (A), (B), and (C), the agent slot is bound to 'person'. After the pistol is introduced the 'AGENT' slot is suddenly re-bound with 'a male person' (D). The 'ACTION' slot undergoes several changes, from 'eat' through 'read' and 'hide' to 'shoot', its final value. An 'INSTRUMENT' slot is introduced after the gun is mentioned in story line three, and the 'GOAL' slot changes repeatedly, sometimes looking like this:

```
F)      ACTION: shoot
        AGENT: a man
        OBJECT: a pile of objects
        INSTRUMENT: a gun
        GOAL: ?
```

The models change repeatedly because there is no particular conclusive way in which candidate models may be evaluated. While some of the models proposed lack plausibility, they could be acceptable accounts of the events in the story if the goal of the actor were known. For instance, if the story had contained the information that the actor wanted to do a ballistics test, then shooting into the given pile of objects might be considered a singularly odd way of conducting such a test, but the model - that the person is shooting the book to 'catch' the bullet - could be evaluated in terms of the information, and other models rejected out of hand.

The model will be further discussed in Chapter 7.

6.3 THE BULLETHOLE PROBLEM AS AN ABSTRACT TASK (1).

The 'BULLETHOLE' problem statement contains statements designed to trigger either real world knowledge or knowledge about SOLO programming concepts. The assumption motivating presenting problems in terms of ordinary, real world events is that such a presentation makes it a lot easier for students to understand what the problem is. Put another way, the assumption is that abstract descriptions of the problem would make the problems harder to solve. There is evidence from studies in deductive and inductive reasoning (Wason, 1978; Wason & St. John Evans, 1975) that people are able to solve problems couched in terms of familiar activities but are unable to solve the same problems when they are presented abstractly, or have a great deal more difficulty with such presentations.

That is the assumption. But perhaps couching programming problems in terms of real world events is a bad idea. It may be that a priori models of the world preempt processes that might otherwise be used to construct an adequate representation of the problem, or lead most students to concentrate on aspects of the problem which are irrelevant to the problem's solution. The majority may be better off with abstract forms of problems. In order to test this possibility, two different abstract forms of the 'BULLETHOLE' problem were devised and the results compared with the results from studies of Subjects solving the original version of the problem. These studies are discussed in the next two Sections.

METHOD. Students at the D303 Summer School (July, 1982) who had selected Artificial Intelligence as their first choice of a Summer School Project were told that the experimenter needed nine Subjects for an experiment and asked for volunteers. The first nine volunteers were accepted as Subjects. All Subjects were tested in the second day of their Project work. All the Subjects had been involved in the AI 'trailer' sessions in which various example programs were run for them, and which they were allowed to 'play around with' in order to determine the kind of modelling project they wanted to attempt. Students work together in small groups at Open University Summer Schools and all the Subjects had been involved in discussions with other members of their groups about the project they would carry out. The AI Project lasts two and a half days, so the students were nearly half way through their Project before they were tested as Subjects in this experiment. Subjects were tested in a small, quiet room. They were given a copy of the problem statement (see Figure 6.4), the SOLO Programming Manual, and several sheets of paper and a ballpoint pen. They were told they could use the Programming Manual at any time, if they wished, and were asked to read the problem text aloud, one line at a time, and to 'think out loud' while they solved the problem contained in the problem statement.

Their comments were recorded on audio cassette tapes and transcribed at a later date.

On page 80 of units 3-4 we looked at a method for making a particular inference 'Keep on happening'.

Suppose SOLO had the following descriptions stored in its database:

```
A----R----B

B----R----C

C----R----D
```

Could you write a procedure which makes the following inference: If a node (for example, A in the database above) is linked to another node (say B) through the relation 'R' then that node (A) should have the description 'X----Y' added to it. Moreover, any node with which an example node is associated should also have the description added to it, and so on.

Thus, for the given database the description 'X----Y' should be added to each node (A,B,C and D), so that the database would look like this after you have run your procedure:

```
A----R----B
!
!----X----Y

B----R----C
!
!----X----Y

C----R----D
!
!----X----Y

D----X----Y
```

You can call your procedure 'ADDON'.

FIGURE 6.4. An abstract version of the 'BULLETHOLE' problem statement.

RESULTS: Of the nine Subjects, only six were able to produce a program. Three typical programs are provided in Figure 6.5. As can be seen, none of the programs would achieve the required result. Indeed, it would be very difficult to say, given the programs, but not the problem, what exactly the programs were meant to achieve.

```
1)                              A-R-B

                        CHECK A----R----?
                              A----R----B


2)                      A----R----B
                           then
                        A----X----Y
                        C----R----B  B----X----Y
                        C----X----Y  C----X----Y

                        TO ADDON
                        [If (X)----R----(Z)
                        Then (X)----X----Y]

                        [If X is] ((X))----R----(squiggle)
                        CHECK (X)----X----Y
                        If present (Z)----X----Y
                        If not (X)----X----Y

                        CHECK (A)----R----B


3)                      TO ADDON /TOP/
                        1 FOR EACH CASE OF /TOP/ R ?
                          [NOTE /TOP/--->X--->Y]
                        1A NOTE /TOP/ X Y
                        1B CHECK [? R] ADDON ?
                        [NEXT]
                        DONE
```

Figure 6.5. Example programs designed by Subjects given an abstract version of the 'BULLETHOLE' problem.

In Chapter 7 there will be a discussion of think-aloud protocols taken from two of six Subjects who worked on the 'BULLETHOLE' problem in our laboratory. For the moment, it is enough to know that of these six Subjects (S5 through S10), four satisfactorily solved the problem by the criterion of producing a program which succeeded in changing the input database to the required output database. The difference between the Subjects who were given the 'concrete' version of the 'BULLETHOLE' problem (S5-S10) and those who were given the abstract version is highly significant (Fisher's exact probability = .010).

Nevertheless, it was thought unsafe to draw any general conclusions from the experiment because there is some reason to doubt that the problem statement make the nature of the problem as clear as it could have been. Although some Subjects seemed to have no difficulty understanding the problem (as opposed to solving it), others clearly did. Figure 6.6 represents a brief extract from the protocol of a Subject who understood the problem, and was attempting to make the problem concrete, and Figure 6.7 gives a brief extract from the protocol of a different Subject who may have been utterly confused simply by the wording of the problem statement.

E: Okay what do you know?

> That's three descriptions, um.... A----R----B, B----R----C, C----R----D. Well these are just code letters, I could translate these into words and make more sense of them.

E: How could you do that?

> Well I would, well where it's got A----R----B I could say Alan, R is the same every time so I could say Alan read er, list B.... No I couldn't I'd have to assume that these were people so I could say Alan read, um.... Alan read to Billy

and then on the next line we've got Billy read to Charles and then Charles read to David. Er, the R is the same every time so it must be an action like read to or call it what you like. And the other things would only make sense if they became objects. Shall I carry on.

FIGURE 6.6. Statements from a protocol given by one of the Subjects performing on the 'abstract' version of the 'BULLETHOLE' problem.

Well I'm reading it to see if there's something I have not read properly because I'm.... I'm checking out whether any information is being given as to what 'X----Y' is, or whether it's purely an abstract relationship that I'm supposed to make inferences about as well. Now I don't understand what I've got to do, I really don't. I don't know what's expected of me.

FIGURE 6.7. Statements from a protocol given by another Subject performing the 'abstract' version of the 'BULLETHOLE' problem.

To circumvent any possible artifacts introduced by expressing the abstract problem in the manner of Figure 6.1, (i.e., directly analogous to the original 'BULLETHOLE' problem) a second version was devised, as described in section 6.4.

6.4 THE 'BULLETHOLE' PROBLEM AS AN ABSTRACT TASK (2).

METHOD: Six D303 students were presented with the version of the problem statement given in Figure 6.8. The purpose of the experiment was to make the statement of the problem as concise and clear as possible. So as not to interfere with the Subjects in any way, protocols were not taken. Subjects were simply asked to read the problem statement and to perform the task required of them. Here is the problem statement:

Given a database describing a relationship between a sequence of nodes, as follows:

```
          R            R            R            R
A---------->B---------->C---------->D---------->E
```

write a program which would result in adding the descriptions

```
        S                S
'A---------->Q, B---------->Q, etc., so that the database finally
```

looks like this:

```
          R            R            R            R
A---------->B---------->C---------->D---------->E
  |            |            |            |            |
  |            |            |            |            |
S |          S |          S |          S |          S |
  |            |            |            |            |
  |            |            |            |            |
  |---> Q <---|<-----------|<-----------|<-----------|
```

Call the program FOO, and use one parameter, i.e., TO FOO /X/.

You should be able to achieve the final result by typing in:

FOO A.

WRITE YOUR PROGRAM IN THE SPACE BELOW, PLEASE.

Thanks for taking part.

FIGURE 6.8. Instructions given to Subjects performing on the 'clear' abstract version of the 'BULLETHOLE' problem.

RESULTS: None of the Subjects solved the 'clearer' abstract version of the problem. The programs produced by the Subjects are given in Figure 6.9, below. Again the difference is significant (Fisher's exact probability =

```
A)
TO FOO /X/
1 NOTE /X/  S  ?
2 CHECK /X/  R  ?
2A If Present: NOTE *  S  ?; EXIT
2B If Absent:  EXIT
DONE

B)
TO FOO /X/
1 NOTE /X/  S  ?B
2 FOR EACH CASE OF /X/  R  *
2A FOO *
DONE

C)
TO FOO /X/
1 CHECK /X/  R  ?1
1A If Present: CONTINUE
1B If Absent;  EXIT
2 NOTE /X/  S  ?1; CONTINUE
3 CHECK ?1  R  ?2
3A If Present: CONTINUE
3B If Absent:  EXIT
ETC.

D)
1 CHECK /X/  R  ?
1A If Present: CONTINUE
1B If Absent:  ----
2 CHECK /X/  S  ?
2A If Present: *  R  ?
2B If Absent:  ---
3 continue like this depending on sequence thrown up by computer until
worked through the database.

E)
CHECK /X/  R  /*/
if present: NOTE /X/  R  Y; CONTINUE
if absent:  FORGET /X/  R  /*/; CONTINUE
CHECK E  FOLLOWS  D
if present: NOTE E  R  Y
if absent:  EXIT

F)
TO FOO /X/
CHECK /X/  R  /Z/
If Present: NOTE /X/  S  /Z/
If Absent:  EXIT
```

FIGURE 6.9. Programs written by Subjects given the 'clear' version of the abstract 'BULLETHOLE' problem.

DISCUSSION: Although no one solved the 'clearer' version of the abstract problem, the solutions actually produced all look to be closer to a correct solution than any of the programs produced under the 'obscure' abstract version of the problem. In fact, one of the solutions (the one marked (B), above) was very nearly correct, albeit for the wrong reason. Solution (B) has a bug in the first step of the program: the instruction should read 'NOTE /X/ S Q'. If we assume for the moment that this particular bug is the result of a temporary slip of attention, then the program would have worked. The solution is given in (D) is interesting in that although it was not a very coherent design, and contained no recursion segment, the Subject's notion of recursion is expressed in the comment about the further behaviour of the incomplete progam. Although the solving process was not recorded for the Subjects given the 'clear' version of the problem, they all indicated to the experimenter that they were unsure what exactly was required of them.

Taken together, the experiments suggest that complex interactions would be expected to occur when the different elements are combined into one programming problem statement. The first study showed that the 'story' elements, presented alone, reliably triggered certain real world knowledge, and that the story was treated as a problem to be solved in terms of the possible goal of the character described in the story. In the first experiment of the second study, Subjects attempted to anchor the elements of the abstract problem in their knowledge of the world, and failing to do so, failed to understand what the problem was all about. It seems that only very low level programming Knowledge was triggered by the elements of the problem statement. In fact, it looks as though the experimental Subjects tried to solve the problem by copying down the triples mentioned in the problem statement in an effort to create a program simply by finding a reasonable pattern for laying the triples out. In the 'clearer' version of the abstract problem, the experimental Subjects achieved more coherent program designs. This problem statement not only was clearer, but arguably presents the problem in somewhat more concrete terms. Perhaps this concreteness helped the Subjects to anchor the problem in other concrete models they possessed, which in turn acted as triggers to higher level programming Knowledge than was possible with the 'hard' version of the problem. This is pure speculation, of course, because, as has been indicated already, protocols were not gathered from these latter Subjects.

In the next Chapter we will turn to an interpretation of the interactions of the two sources of Knowledge when the protocols given by Subjects S5 and S8 are studied in some considerable detail.

CHAPTER 7

A THEORY OF PROBLEM SOLVING IN COMPUTER PROGRAMMING.

7.1 INTRODUCTION

In Chapter 6 a model of story understanding processes was outlined. In this Chapter that model will be moulded into a theory of problem solving processes in the particular domain of programming. The theory applies to both novice and expert programmers. The major categories of behavior postulated by the theory are these:

1) Reading.
2) Schema activation.
3) Hypothesis generation, and
4) Confirmation or Disconfirmation of Hypotheses.

That is, the very first stage of problem solving in programming is bottom up, involving reading a part of the problem statement. However, the theory predicts that programming knowledge will be activated by features of the problem statement, and that this activated knowledge takes a leading role in guiding the search for information in the problem statement. This active knowledge achieves its effect by serving as the basis for hypotheses about the contents of the problem, and these hypotheses may be either confirmed or disconfirmed by that information content. If, and when, all of the hypotheses from activated programming concepts are satisfied, the problem is solved. The theory is an 'interactionist' theory in the sense that understanding and solution processes are posited to operate together. That is, solution processes are not isolated from the problem understanding process, as in the 'UNDERSTAND' system described in Chapter 2. (See Chi, Feltovich & Glaser for a model of interactive processes in understanding/solving physics problems, and McDermott & Larkin for a 'stage' (non-interacting) model.) It is interactionist also in the sense that different sources of knowledge contribute to the understanding processes. Programming knowledge makes an important contribution by directing attention to important aspects of a problem description. As a simple example, the 'BULLETHOLE' problem statement - as indicated in Chapter 6 - contains some statements designed to trigger programming knowledge, and some statements designed to trigger world knowledge. In terms of the program that must be designed as a successful solution to the 'BULLETHOLE' problem, some aspects of the real-world knowledge that would be triggered are irrelevant, and other aspects crucial. An understanding of the layout of the objects mentioned in the problem statement, and the operation performed on them is important. Understanding the goal of the person described in the story is not important. The theory may best be explained by illustration. Figure 7.1 shows a schema-like representation of the knowledge a 'competent' novice might have about recursion, including coding of recursive procedures. The 'competent' novice would be one who had, say, the Loop model of recursion, i.e., has a model of the behaviour of recursive procedures, and is able to mentally simulate the behaviour of procedures of that type (perhaps inaccurately; see Chapter 4)

```
$RECURSION
GOAL: (ForEvery x In (my applies-to) do
        (achieve (my action) x)
ACTION: (a side-effect {DEFAULT (a NOTE) })
APPLIES-TO: (a chain-relation list).
LOOP-FOR: (ForEvery x In (my applies-to))
TRIGGER: 'Keep on happening', 'reapply'
SURFACE-TEMPLATE:
      TO (name1 =(a name)) (a parameter {default: X})
        (my action)
      CHECK (a variable) (a relation) (a wildcard)
      IF PRESENT: (a procedure
                       with name = name1
                       with parameter = *); EXIT
      IF ABSENT: EXIT
EVALUATION-RULES:
 1) (let parameter = the startnode from (my applies-to))
 2) (apply (my action) parameter)
 3) (assert '(ACHIEVED ,(my action) ,parameter))
 4) (let parameter = (GetNextNode))
 5) (ForEvery x In (GetRestOfNodes)
              (assert '(ACHIEVED ,(my action) ,x)))
```

FIGURE 7.1. A Recursion schema.

A schema-like representation of the chain-relation list schema referred to in the Recursion schema is given in Figure 7.2.

```
CHAIN-RELATION-LIST
SERVANT-OF: (Recursion)
SURFACE-STRUCTURE:
node1---chain-relation---node2---chain-relation---node3---etc.
SURFACE-TEMPLATE:
      (a chain
             with node1 = (a node)
             with relation = (a chain-relation)
             with successor-nodes = (a list of nodes))
```

FIGURE 7.2. A chain schema.

The Recursion schema encodes the following information. The Goal of a recursive procedure is to achieve a particular action on every node in a particular chain of nodes. Implicit in the schema's 'Evaluation-rules' is the belief that recursive procedures are meant not simply to achieve an action on every node in a particular list, but to do so in the order in which they are encountered, i.e., from the front to the end of the list. The index to the schema is one of two key words or phrases: 'reapply', or 'Keep on happening' (terminology repeatedly used in the SOLO Programming Manual in the discussion of recursion). The Action slot has a default specification that the action to be performed will be a side-effect on the database (NOTE = addition to the database). A particular database structure is specified in the 'Applies-to' slot: a chain-relation list. The schema also contains a 'surface-structure' representation of an idealized recursive procedure, and the 'Evaluation-rules' embody the Loop model of recursion described in chapter 4.

Assume that this schema was triggered by the Keywords 'Keep on happening' in the first line of the problem statement. As discussed in Chapter 6, instantiation of the schema is accomplished by binding values to its various slots. One way to think about this 'binding' operation is to suppose that each slot's specifications serve as the basis of hypotheses about the information contained in the problem statement. So, the 'hypotheses' arising from the recursion schema can be informally stated as:

Hypothesis-1: the problem has to do with a list of objects.

Hypothesis-2: the objects will be linked by a chain relation.

Hypothesis-3: some action will be specified.

Hypothesis-4: the action to be performed will be achieved first on the object at the head of the list, then on all of the other objects in the tail of the list.

These hypotheses represent the Knowledge which is presumed to be encoded in procedures that are attached to the various slots in the recursion schema, and which become active when a copy of the schema is instantiated. As long as the schema is active, its procedures set up searches for specific information to bind the schemas slots to. Default values are instantiated in the case of failure of the attached procedures to produce a result. It is also proposed that schemas can activate other schemas, and are able to use each others' partial and final results (Anderson, 1976). Examples of such processing mechanisms will be discussed in Chapter 8, but not the exact nature of the processes, as these are unknown.

Of course, the 'real world' schemas discussed in Chapter 6 with respect to the 'BULLETHOLE' story also will be triggered by other statements in the problem text. The theory renders the hypotheses from these schemas active but generally inefficacious by the following rule: the hypotheses from activated domain related knowledge have priority in guiding the search for information in a problem description; call such a domain related schema a 'predominating' schema. If domain related knowledge is not triggered, or is 'content free', then the real world schemas would take a predominating role in guiding processing. By 'content free' I mean that some novices are presumed to have acquired the names of concepts, but not much more. Such a schema might look something like this:

RECURSION
GOAL: ?
ACTION: ?
TRIGGER: 'Keep on happening'.
EXAMPLE: 'INFECT' procedure in Programming Manual.

Several predictions are made from the theory. First, given a problem statement that contains a pointer to the Recursion schema, that schema will be activated and guide the search for information in a problem statement. There is such a pointer in the first statement of the problem text: "On page 80 of Units 3-4 we looked at a method for making a particular inference 'Keep on happening'". [NOTE: Units 3-4 = SOLO Programming Manual]. Second, the important aspects of the problem statement in the programming task are those that have to do with the layout of objects and the action performed, and not the motive of the person performing the action. That is, given the following schema-like representation of a Shooting schema and subschemata:

```
SHOOTING
GOAL: (the Goal in (my SPECIALIZATION))
AGENT: (a Person {DEFAULT (a Man)})
INSTRUMENT: (a gun {DEFAULT (a Pistol)})
OBJECT: (the Object in (my SPECIALIZATION))
CONSEQUENCE: (the Object in (my SPECIALIZATION) has a
                      bullet hole)
           (the Agent ...
SPECIALIZATION: (an Event (of type: Aggression))
SPECIALIZATION: (an Event (of type: Sport))
SPECIALIZATION: (an Event (of type: Hunting))
.....

SHOOTING/Aggression
GOAL: (one from (vengeance, jealousy, etc.))
AGENT: (a Person {DEFAULT (a Man)})
INSTRUMENT: (a gun {DEFAULT (a Pistol)})
OBJECT: (an Animate-Object {DEFAULT (a person)})
CONSEQUENCE: .....
.....

SHOOTING/Sport
GOAL: (achieve Esteem)
AGENT: (a Person {DEFAULT (a Man)})
OBJECT: (an Object {DEFAULT (a Target)})
INSTRUMENT: (a gun)

.....

SHOOTING/Hunting
  GOAL: (one from {..... ...})
AGENT: (a Person {DEFAULT (a Man)})
OBJECT: (an Animate-Object {DEFAULT (an Animal)})
INSTRUMENT: (a gun)
.....
```

it is predicted that when the Shooting schema is triggered in the programming task, the Sport track (a specialization of the more generic

Shooting schema (see Schank & Abelson, 1978)) would be selected as a candidate schema for understanding information gained before the point at which the Shooting schema was activated, and for making further predictions about needed other information. In the story understanding task, one of the four Subjects, on reading the relevant statement from the story, said: "I don't know what to make of that." In their first response to the statement, the other three Subjects first mentioned:

1) something sinister about the scene,

2) a hide-out,

3) a spy.

The responses all indicate that the Aggression track had been chosen to combine with other knowledge to form a model of the events that had been described in the story. The Subject whose complete protocol was analyzed in Chapter 6 first of all thought of a hide-out, and subsequently thought of Shooting-range (Sport track of the Shooting schema) but dismissed the idea with the statement: "But hardly with a sandwich on it". In Chapter 8 we shall see that these predictions were only partially supported.

7.2 AN INTERPRETATION THEORY FOR SCORING VERBAL PROTOCOLS.

The aim of this section of the Chapter is to provide a formal statement of the problem solving strategy in terms of an interpretation theory for the analysis of verbal protocols. The theory indicates the patterned behavior of problem solvers when they are confronted with a problem statement.

There are two major approaches to protocol analysis (Breuker, 1981). One approach is bottom up in that it involves searching the protocol for information, and implies a fair amount of ignorance about what information the protocol contains - before the search begins, at any rate. Examples of this kind of analysis can be found in Chapter 4, where the comments on the Program Questionnaires were used to make inferences about respondents' models of recursion. The model of problem understanding developed by Hayes & Simon was a data driven model: they went to the protocol to find out what was in it, and created their model to account for what they found. (See Byrne, 1976, for a discussion of the good uses to which such analyses may be put.)

The second approach is top down: one begins with a good idea of what the protocol contains and uses the protocol to confirm or disconfirm expectations. In such cases, protocols can be considered as tests of a model or a theory. The behaviour recorded in the protocol should closely match that predicted by the theory if the theory is to receive support.

It was indicated in the first section of this Chapter that the major categories of behaviour postulated by the theory presented here are these:

1) Reading.
2) Schema activation.
3) Hypothesis generation, and
4) Confirmation or Disconfirmation of Hypotheses.

Rules are given below for classifying protocol statements as instances of one or another of these categories of behaviour.

In addition to these four categories, there are several other, less important categories, which are needed to account for the subjects protocol in detail. These different behavior categories, and rules for scoring lines of protocol in terms of them, are formally specified below. The terminology differs slightly from that of Chapter 6 because it is easier to describe the problem solving processes discussed in the next chapter in terms of the activation and interaction of 'schemas' rather than 'models'.

1) READING: A person reads a line from the problem text. RE-READING also occurs in the course of problem solving, and is assumed to be related to search and integration processes.

2) SCHEMA ACTIVATION: Schemas are activated in one of two ways. First, they may be triggered by elements of the problem statement, i.e., they are a response to READING, or RE-READING, and a line of protocol immediately subsequent to these processes should be examined for indications of this process. Schema activation of this type is denoted on the analyzed protocols as 'S-A(C)' which stands for 'Cued Activation of a Schema'. Activation of a schema is often signalled by comments such as "Oh, that reminds me of (something or other)". Concepts may also be associatively cued, through processes of spreading activation (Collins & Loftus, 1972) or associative retrieval processes, (Simon, 1979). On the analyzed protocols, 'S-A(A)' stands for 'Associative Activation of a Schema'. An example of 'cued' and then 'associative'

activation is this. Suppose someone read the second line of the problem statement: "Further, this world is highly organized: the sandwich is on the plate, which is on the newspaper, which is on the book, which rests on the table, which, of course, is sitting on the floor". If the Subject responded: "That's a funny sort of house", then 'a funny sort of house' would be an instance of cued activation of a concept. If the Subject then went on to say: "Reminds me of my mother", then 'mother' would be an instance of an associatively cued schema, his concept of his 'mother' being somehow associatively linked with his concept of 'funny sorts of houses'. All mentions of SOLO constructs, and most programming concepts should be marked as instances of schema activation, if there is no mention of these in the actual problem statement. If the Subject had originally responded: "This reminds me of For Each Case Of" then 'For Each Case Of' would also be an instance of cued activation of a schema. The discussion has focussed on the activation of programming knowledge, but these comments, and those to follow, apply also to other knowledge. Activation of a schema is equivalent to making the information encoded in the schema available for problem solving.

3) HYPOTHESIS-GENERATION: The hypotheses from activated schemas are often reflected in statements of expectations. In general, HYPOTHESIS-GENERATION may be recognized as some sort of question. For example, one of our Subjects, upon reading Line 4 of the problem statement ("Imagine somebody standing beside the table with a .357 Magnum pistol.), said: "Is he going to shoot the sandwich or something?" The line would be scored as HYPOTHESIS-GENERATION.

4) CONFIRMATION & DISCONFIRMATION: When the person sets up an hypo- thesis, subsequent lines of the problem statement (or trains of thought) may provide evidence for or against the hypothesis. Information in problem statements read previously may confirm or disconfirm hypotheses, but unless the reader has kept all previously acquired information in mind, then RE-READING would be necessary in order to discover it. CONFIRMATION is reflected in statements such as the following: "Ah, I was wondering about that...." "That clears up what I said before...." And so forth.

5) PROBE: Knowledge may be more or less well organized, more or less complete. In order to determine whether the knowledge one has is appropriate to a task in hand, one sometimes needs to critically examine that knowledge. Examinations of knowledge are called PROBEs. They would be reflected in a protocol where a person explains what he knows about a concept, or apparently defines a concept, or considers its relevance for the task in hand.

6) ELABORATION: When a person uses an activated schema to organize material, he is said to elaborate the material. For example, when given the second line of the 'BULLETHOLE' problem, some Subjects elaborate the line - which only mentions six objects without specifying relations between them, or any thing else about them - in terms of the 'dining-room' schema. ELABORATIONs add information which was not specifically stated in the problem statement.

7) DESIGN: A person may respond to a portion of text by discussing implementation details in general terms. If a person says something

like: "It looks as though I might need a subprocedure for this.", or, "I'm just going to ignore the details for the time being.", then the person is obviously concerned with program design at a very general level.

8) M-CODING: The 'M' in M-CODING designates 'mental'. This occurs when a person attempts to write a program 'in the head'; without access to a terminal or attempting to write down a program. This process is distinguishable from DESIGN in the level of detail; M-CODING is an implementation of a design.

9) EVALUATION: This activity takes several forms. EVALUATION occurs when a person determines the output for a segment of code he has written. Also, EVALUATION occurs when a person attempts to determine the appropriateness of a particular algorithm for the current problem.

10) PROCESS-COMMENT: The person comments on his own mental processes or states of knowledge. Here are some examples: "That makes sense". "This problem's too hard". "I can't see what I'm meant to do".

11) EXPERIMENTER-COMMENT: Any comment made by the experimenter. The experimenter is indicated by 'E:' at the front of a line of protocol.

12) RESPONSE/CLARIFICATION/QUERY: Here are lumped together three types of behaviour. In general they have to do with the working relationship between the Subject and the Experimenter, rather than with problem solving per se. For example, the experimenter may know that a

particular Subject is a slow typist and offer to type in a database, or may offer the Subject a cup of coffee, and so on. The Subject may be unsure about procedure, e.g., should he keep talking even when he is typing a program at the terminal, or, is he allowed to make notes, etc. These will be demonstrated below.

7.3 THE BULLETHOLE PROBLEM: S8'S PROTOCOL (READING PHASE)

This section presents a detailed analysis of the reading phase of the protocol taken from S8 while the Subject worked on the 'BULLETHOLE' problem. The full text of the 'BULLETHOLE' problem is presented immediately below, followed in the next section by the protocol itself, and then by a detailed discussion of the important characteristics of the protocol.

7.3.1 THE BULLETHOLE PROBLEM STATEMENT

On page 80 of Units 3-4 we looked at a method for making a particular inference 'keep on happening'.

In this option you are asked to imagine a state of the world in which there are six objects: a sandwich, a plate, a newspaper, a book, a table and a floor.

Further, this hypothetical world is highly structured: the sandwich is lying in the centre of the plate, which is sitting on the newspaper, which is lying on the book, which is on the table, which of course rests on the floor.

The objects and their relationships can be represented as the following:

```
SANDWICH...ON...PLATE

PLATE...ON...NEWSPAPER

NEWSPAPER...ON...BOOK

BOOK...ON...TABLE

TABLE...ON...FLOOR
```

Now imagine someone standing beside the table with a .357 magnum pistol.

The person puts the muzzle of the gun on top of the sandwich, so the gun is thus pointed towards the floor.

A .357 magnum will, when fired, put a hole in anything in the path of the bullet.

The person fires the pistol.

Can you write a procedure which makes the following inference: when you shoot an object that object has a bullethole; moreover, any object which it is on also has a bullethole, and so on.

Your procedure, and its effect on our database might look something like this:

```
SOLO:   SHOOTUP SANDWICH

      SANDWICH...ON...PLATE
        !
        !...HAS...BULLETHOLE

      PLATE...ON...NEWSPAPER
        !
        !...HAS...BULLETHOLE

      NEWSPAPER...ON...BOOK
        !
        !...HAS...BULLETHOLE

      BOOK...ON...TABLE
        !
        !...HAS...BULLETHOLE

      TABLE...ON...FLOOR
        !
        !...HAS...BULLETHOLE

      FLOOR...HAS...BULLETHOLE
```

In other words, you are asked to write a procedure which 'puts bulletholes in objects' that are stacked up like these, starting with the object on top.

7.3.2 S8'S PROTOCOL (READING PHASE)

The segment of protocol presented here - one hundred lines - includes the Subject's comments only during the reading of the problem statement, from presentation of the problem until the last line has been read and commented on by the Subject. This part of the protocol is given in its entirety, and discussed in terms of its fit to the pattern of behaviour predicted by the interpretation theory. Here, then, is the protocol:

1 "On page 80 of Units 3 to 4 we looked at a method for making a particular inference 'keep on happening'.	READING
2 Um, I can't, um....	META-COMMENT
3 Is that called 'iteration'?	S-A(C)
4 No, 'recursion'.	DISCONFIRMATION/S-A(C)
5 I can, I can remember something about being told something about the distinction between iteration and recursion and one goes sort of like along a database and the other sort going down.	S-A(A)
6 Well, that's how I sort of thought of it	META-COMMENT
7 Well....	?
E: Alright, any further expectations or any....	EXP-COMMENT
8 Well....	?
E: I just want to know everything you uh, I mean as you read it....	EXP-COMMENT
9 So, I think this is going to say something about what happens when you keep on applying a function through a database.	HYPOTHESIS
E: Okay	EXP-COMMENT
10 Is that okay, can I go on?	QUERY
E: Yeah.	EXP-COMMENT
11 "In this option you are asked to imagine a state of the world in which there are six objects: a sandwich, a plate, a newspaper, a book, a table and a floor."	READING
12 Um, so, it looks like I've got to just imagine those things in isolation and then just going to be told something about them which will relate them or something...	META-COMMENT/ HYPOTHESIS
13 (So) I make sense of that.	META-COMMENT
14 I find it very difficult to imagine things....	META-COMMENT

15 I mean you've got to put them in some sort of sensible organization... META-COMMENT

16 So, right at the moment I could be imagining a room with a floor, with a sort of quite distinctive pattern.... S-A(C)

17 So it, so that it really looks like a floor to me. META-COMMENT

18 And table on which there's a book, a newspaper and a plate with a sandwich on it. HYPOTHESIS

19 I mean that's how I imagine it now but I... META-COMMENT

20 Perhaps I'm going to be told something to make me alter that. HYPOTHESIS

21 Allright. ?

22 "Further this hypothetical world is highly structured: the sandwich is lying in the centre of the plate, which is sitting on the newspaper, which is lying on the book which is on the table which of course rests on the floor." READING

23 Well that reminds me of the first bit where it talks about 'Keep on happening'.... CONFIRMATION

24 'Cause it's like uh, the function keeps on happening, it's like the function, or like NOTE something is on, is on something else, which you could keep applying. S-A(A/A)/ HYPOTHESIS

E: Okay. Anything else? EXP-COMMENT

25 Um, yeah 'cause, well you could also get out things like the.... HYPOTHESIS

26 The sandwich is on top of the floor but the newspaper is under the sandwich sort of thing, even though they're not directly.... HYPOTHESIS

27 By going through that in stages and stoppingand sort of making inferences about if the sandwich is on the plate which is on the newspaper, the sandwich is on the newspaper. HYPOTHESIS

E: Okay. EXP-COMMENT

28 Right. ?

29 "A database representing this state of affairs looks like this...." READING

30 Keep going? Yeah. QUERY

E: You can look at the entire database.	EXP-COMMENT
31 "Sandwich on plate, plate on newspaper, newspaper on book, book on table, table on floor."	READING
32 Is that it? Yeah.	QUERY
33 Yeah. Well, that's sort of like you'd expect, I mean....	CONFIRMATION
E: Yeah. So in fact you predicted that exactly....	EXP-COMMENT
34 Yeah, because I mean I'm obviously reading this in the context of thinking about databases and that and that's how I'd imagine it from what I was saying so that just confirms that that was okay,	META-COMMENT
35 But, uhmm....	?
36 I mean, course plate, they'd all be joined up wouldn't they because plate is the same node isn't it....	PROBE
E: Yeah, that's the way they showed it....	EXP-COMMENT
37 Yeah, yeah, I know, but that's how I like to think of it....	META-COMMENT
38 Right. "Now imagine someone standing beside the table with a point... 357 magnum pistol."	READING
39 Is he going to shoot the sandwich or something?	HYPOTHESIS
E: Why do you think that?	EXP-COMMENT
40 I don't know, I mean....	META-COMMENT
E: You said it. I'd like an explanation for it.	EXP-COMMENT
41 I just had this sort of picture of standing, standing by the table with this gun....	S-A(C)
42 So I thought it must be pointing at something....	HYPOTHESIS
43 And that something....	HYPOTHESIS
44 Well, the sandwich is on the top, and....	HYPOTHESIS
E: And if that were the case what would happen, what are your expectations?	EXP-COMMENT

45 Well, I sort of kind of imagined shooting the sandwich, the bullet going into it, sandwich all over the place and smashing the plate and smashing the table and....	HYPOTHESIS
46 I don't know very much about point 357 magnum pistols. I don't know how far the bullets go....	HYPOTHESIS
E: I see. Anything else?	EXP-COMMENT
47 Uh, no.	RESPONSE
E: Okay. Carry on.	EXP-COMMENT
48 "The person puts the muzzle of the gun on top of the sandwich, so the gun is thus pointed towards the floor."	READING
49 Well, I would expect him to shoot through all that lot then.	HYPOTHESIS
E: You expect him to shoot through all that lot?	EXP-COMMENT
50 Yeah. Because it's pointed towards the floor it's got to go through the others.	HYPOTHESIS/PROBE
E: Okay.	EXP-COMMENT
51 I don't know why he wants to do it though.	HYPOTHESIS
52 But that's it really.	META-COMMENT
E: Okay.	EXP-COMMENT
53 "A point 357 magnum will, when fired, put a hole in anything in the path of the bullet."	READING
54 Oh, that's what I was wondering about before.	CONFIRMATION
55 So it looks you're gonna get a hole in your sandwich, your plate, your newspaper, your book, your table and your floor, and anybody underneath in the room below.	HYPOTHESIS
56 Right....	?
E: Anything else?	EXP-COMMENT
57 I mean, that's quite a simple sort of thing....	META-COMMENT
58 Except, I mean, sort of makes you think	CONFIRMATION/

that, that they must be powerful sorts of guns or whatever....	ELABORATION
E: Must be what?	E-COMMENT
59 Powerful guns, yeah?	RESPONSE
E: It's extremely powerful, yes.	E-COMMENT
60 Right.	RESPONSE
E: You could knock down an elephant at five thousand yards....	E-COMMENT
61 Yeah? Must be what was in the film.	RESPONSE
E: Anything else? -- Uh, Dirty Harry? Have you seen it?	E-COMMENT
62 No, I was thinking about 'Gloria' last night, there was a magnum in that.	RESPONSE
E: Oh, well, that was probably a 357.	E-COMMENT
63 Can I go on?	QUERY
E: Nothing else? Yeah. Give me that (first sheet) page, and uh.... Well here, in fact you get to keep that. Because once you've read a line if you ever want to go back you obviously can.	EXP-COMMENT
64 Okay. I mean I was sort of looking back and....	RESPONSE
E: Tell me when you do that.	EXP-COMMENT
65 Right. Okay. Well, I was looking back when it said 'put a hole in anything in the path of the bullet' I was obviously going through that and reading 'sandwich etc. etc.'.	RE-READING
E: Yeah. Okay.	EXP-COMMENT
66 "The person fires the pistol."	READING
67 So it does get a hole in all them things, yeah?	CONFIRMATION
E: Okay. Anything else.	EXP-COMMENT
68 There's a mess everywhere I should think. I just wonder why he did it.	HYPOTHESIS
69 It seems a bit pointless....	PROBE (FAILURE)

70 Right. Here's another one.	COMMENT
71 "Can you write a procedure which makes the following inference: when you shoot an object that object has a bullethole; moreover any object which it is on also has a bullet hole and so on."	READING
72 Yeah, I should think I can.	META-COMMENT
73 Do you want me to?	QUERY
E: Uh, finish the.... You're confident now you can do it?	EXP-COMMENT
74 Yeah.	RESPONSE
E: Uh, do you know how to do it at this point?	EXP-COMMENT
75 Umm, I think so, yeah, I mean....	META-COMMENT
E: Tell me what it is.	EXP-COMMENT
76 You'd have to do things like....	PROBE
77 Ummm....	?
78 I don't know, like....	META-COMMENT
79 Like TO SHOOT....	M-CODING
80 You've got some function called TO SHOOT....	M-CODING
81 And you, TO SHOOT....	M-CODING
82 What's that, is it a parameter?	QUERY
E: Yeah.	EXP-COMMENT
83 Right. NOTE PARAMETER HASA BULLETHOLE.	M-CODING
84 CHECK PARAMETER ON QUESTION MARK.	M-CODING
85 IF PRESENT NOTE ASTERISK HAS BULLETHOLE....	M-CODING
86 No, you put IF PRESENT SHOOT, no, you put ASTERISK then don't you, because that's a whatsit, a thing in a box....	M-CODING/ S-A(A)
E: Okay. Any other thoughts?	EXP-COMMENT
87 Ummm....No, I mean apart from the function I haven't put EXITs and CONTINUEs and things like that.	EVALUATION
E: So do you think that at that point you	EXP-COMMENT

would've worked it all out before going on? Or just had confidence that you could do?	
88 Ummm.... I think I'd have had a go because I do things like that 'cause like when I was reading through that I kept thinking of things:	META-COMMENT
E: Well, how would you do that?	EXP-COMMENT
89(And sort of something that I'd scribble)???	?
90 And all the ones, I mean all the SAQ's and also the ones that it just says "try this out" and then you get the answer the next line, I mean I actually covered the answers up and did all those, 'cause I find it really helps to understand what's going on, no?	CLARIFICATION
E: Okay. Anything else?	EXP-COMMENT
91 No.	RESPONSE
E: Okay. Go on.	EXP-COMMENT
92 "The procedure and its effect on our database might look something like this:"	READING
93 So I read all this?	QUERY
E: Yeah.	EXP-COMMENT
94 "Shootup sandwich. Sandwich on plate. Sandwich has bullethole."	READING
E: Read that whole page, because, in fact, it continues to the other page to some extent, so let's get to the end of the database.	EXP-COMMENT
95 Yeah. I mean that's what, that's what SOLO puts out after you tell it what to do next.	HYPOTHESIS
E: Right.	EXP-COMMENT
96 SHOOTUP....SANDWICH. Right. (reads all of database silently)	READING
97 That's okay. That's what I'd expect if you've got your NOTE things and your CHECKs ON, then NOTE, so you have your HAS all under the ONs, so that's okay....	CONFIRMATION
E: Okay. Last line.	EXP-COMMENT
98 Okay. Is this the last one so I can take this away? (This=covering sheet)	QUERY

E: Yeah.	EXP-COMMENT
99 "In other words you are asked to write a procedure which 'puts bullet holes in objects' that are stacked up like these, starting with the object on top."	READING
100 That's okay. Quite a nice way of describing it.	META-COMMENT

7.3.3 Overview Of The Protocol.

From one perspective the protocol is a trace of the construction and elaboration of an image generated in response to the list of objects given in the second line of the problem statement. A representation of the image first constructed is provided in Figure 7.3. Figure 7.4. represents subsequent alterations of and additions to the image.

```
                         0  = Sandwich
                         |
                         |--ON
                         |
           0  = Book     0  = Plate       0  = Newspaper
           |             |                |
           |--ON         |--ON            |--ON
           |             |                |
           --------------0  = Table--------
                         |
                         |--ON
                         |
                         0  = Floor
              HAS
      0  = Pattern
```

FIGURE 7.3

```
            0 = Man
            |
            |----------IS---STRANGE
            |
            |--HAS
```

```
    |
    0 = Gun
    |
    |----------IS---POWERFUL
    |
POINTED-AT
    |
    |------------0 = Sandwich ----IS--SMASHED
                 |
                 |--ON
                 |
                 0 = Plate -------IS--SMASHED
                 |
                 |--ON
                 |
                 0 = Newspaper ---IS--SMASHED
                 |
                 |--ON
                 |
                 0 = Book --------IS--SMASHED
                 |
                 |--ON
                 |
                 0 = Table -------IS----SOLID
                 |
                 |
                 |----------------RESISTS---PENETRATION
                 |
                 |--ON
                 |
                 0 = Floor
```

FIGURE 7.4

The coding of the protocol indicated the triggering of ten schemas by elements of the problem statement: iteration, recursion, database, room, pattern(floor), function, NOTE, whatsit, thing-in-a-box, shooting.

7.3.4 The Recursion Schema.

In Section 7.1 a recursion schema representing what it was presumed was known by a 'competent' novice, a novice with the Loop model of recursion, was presented in formal, KRL-like notation, and again in an informal statement of the hypotheses inhering in the formal version. In order to make further discussion easy to follow, the informal statement of the hypotheses from recursion will be used from time to time, although the binding of the schemas slots will be presented formally, as and when they are claimed to occur. As stated, the Hypotheses inhering in the various slots are these:

(a) The problem has to do with a list of objects.

(b) The objects will have a particular relationship (probably transitive) with one another.

(c) The problem will mention some action to be performed on all the objects mentioned.

(d) The action to be performed will be achieved first on the object at the head of the list, then on subsequent objects, until all the objects have had the action performed on them.

7.3.5 The First Problem Statement.

Both the Iteration and the Recursion schemas were activated by some trigger in the first line of the problem text - presumably the words 'Keep on happening' (see S8's phraseology at line 9). DISCONFIRMATION of Iteration occured almost the moment it was triggered, although why this is so is not clear. One possibility is that Recursion was associatively activated from Iteration, and its activation served to DISCONFIRM the triggering schema. The combination of Iteration and Recursion associatively trigger a database schema. Whether the Subject knows which schema is related to which form of search (line 5 of the protocol) is also not absolutely clear. However, if the referent in the last phrase of the comment at line 5 is Recursion, then S8 can be presumed to know which is which. Also, the single HYPOTHESIS arising in this Episode of protocol (line 9) sounds like a definition of Recursion: a 'function' which you 'keep on applying'. As such, a copy of the schema is presumed to have been instantiated at this point, and its problem solving processes set to search for data to which its slots could be bound.

7.3.6 The Second Problem Statement.

The schemas that would be expected to be triggered by this line, in terms of the data discussed in the previous Chapter, are Room, Eat, Read, Relax, and/or Personality-type. Once activated, a schema's slot processes are triggered and a search for information is begun. We have seen that the default for the AGENT: slot of each of Eat, Read, and Relax is usually instantiated, as are the GOAL:, ACTION:, and OBJECT: slots. There is no evidence of these schemas being activated in S8's response to this line of the problem text. All attention in this Episode is devoted to constructing an organization for the objects mentioned in the line. The theory argues that the role of domain related knowledge would be to direct attention away from inessentials so there is a small amount of support for the theory here, although there is some concentrated effort by the Subject to focus a mental image of a 'floor with a quite distinctive pattern'.

The Episode begins with an HYPOTHESIS that something will be said which will show the solver how to relate the objects mentioned in the line. Two of the hypotheses from the Recursion schema are 1) that there will be a list of objects, and 2) that there will be a particular type of relation holding between them. The Subject does indicate an organization for the objects (line 18) but it would be stretching a point to argue that a Recursion schema - even one with strong expectations about the relationship between a set of objects, and processes actively searching for one - would need to be active in some one's mind in order for the person to make the particular inference that the 'thought up' organization might not be correct. Binding of Recursion's slots to the objects would be tentative at best, and it is presumed that the Recursion schema is still empty at this point in the processing. The Episode also closes with a restatement of the particular HYPOTHESIS. On balance, the evidence for a predominating role for the Recursion is weak.

7.3.7 The Third Problem Statement.

'Function' (line 24) has been scored as associatively activated through the 'Keep on happening' of which the Subject is reminded by the description of the relations holding between the objects in the third line of the problem text. NOTE (also line 24) is scored as having been associatively cued, in its turn, by 'function'. The first HYPOTHESIS also is derived from line 24: that you could reapply NOTE <something>. The sequence serves as CONFIRMATION for the HYPOTHESIS found in line 9, Episode 1. It may also serve to 'refresh', or rehearse the notion of reapplication of a function. The other HYPOTHESES scored in this Episode (lines 25, 26, and 27) are simply further specifications or elaborations of the first hypothesis of the Episode.

7.3.8 The Fourth Problem Statement.

With the information contained in this line, the first two hypotheses from the Recursion schema:

HYPOTHESIS-1: The problem has to do with a list of objects.
HYPOTHESIS-2: The objects will have a particular relationship (probably transitive) with one another.
are CONFIRMED. The processes attached to the Recursion schema's APPLIES-TO: slot are presumed now to have constructed an appropriate representation of the list of objects, as:

APPLIES-TO: (sandwich ON plate ON newspaper ON book ON table ON floor)

The Recursion schema now looks like this:

```
RECURSION
GOAL: (ForEvery x In (my applies-to) do
        (achieve (my action) x))
EFFECT: (a side-effect {DEFAULT (a NOTE)})
APPLIES-TO: (sandwich ON plate ON newspaper ON book ON table ON floor)
```

There is good evidence for a role for the Chain Relation subschema in the Subject's comments at lines 36 and 37. The problem statement example database represents all but the first and last nodes twice, so that a zigzag trail between SANDWICH and NEWSPAPER is suggested:

SANDWICH ON PLATE

PLATE ON NEWSPAPER

NEWSPAPER ON BOOK

etc.

The pattern (repeat of nodes) is at variance with the surface template slot of the Chain Relation schema:

[chain relation surface template schema]
and the Subject registers a complaint, even though it is understood that the problem representation is equivalent to the Subject's preferred way of thinking about such database structures.

7.3.9 The Fifth Problem Statement.

A strong argument for the role of the Recursion schema will be made in this section. The final two HYPOTHESES from Recursion are these:

HYPOTHESIS-3: some action will be performed on the objects mentioned.

HYPOTHESIS-4: the action to be performed will be achieved first on the object at the head of the list, then on all of the objects in the tail of the list.

We now want to understand S8's comments in lines 39, 42-44, and 45-46 in terms of the interactions of these schemas, with the Recursion schema predominating. Stated very informally, the Recursion schema wants to know if there is some action that can be performed on all the objects in its Chain Relation List, beginning with the object (SANDWICH) at the front of the List and thence with all the other objects in their order of appearance on the list. The APPLIES-TO: slot of the Recursion schema has already been bound to the list of objects given in line 3 of the problem statement (see above) when the restriction on the binding was satisfied by the Chain Relation linking the objects. The HYPOTHESES from the current instantiation of the Recursion schema come from the GOAL: and EFFECT: slots. The relevant slots of the Recursion schema at this point in the processing are these:

```
RECURSION
GOAL: (ForEvery x In (my applies-to) do
          (achieve (my action) x))
EFFECT: (a side-effect (DEFAULT (a NOTE) ))
APPLIES-TO: (sandwich ON plate ON newspaper ON book ON table ON floor)
```

S8's comments in line 39 indicate that a lot of processing has occured immediately after the Subject read the line. The Subject's comments in line 39 can be represented as the filling of some of the slots in the Shooting schema which was outlined in SECTION 7.1, and which is given again below in Figure 7.5. The slots for AGENT:, INSTRUMENT:, and OBJECT: have been filled to reflect S8's comments in the protocol line 39. The AGENT: slot is instantiated with the slot's default value (a Man), the INSTRUMENT: slot with information from the current line of the problem statement (a .357 Magnum pistol), the OBJECT: slot with information from a previous line of the problem statement, and the CONSEQUENCE: slot with the expected effect of shooting an object. The GOAL: slot has not been filled because S8's statement is in the form of a query, suggesting uncertainty that the interpretation is correct. The SPECIALIZATION: slot has been filled with the name of the track through which information is filtered in the creation of the interpretation. What we want to know now is how the filling of these slots came about. We shall discuss this with reference to protocol lines 41-46.

```
SHOOTING
GOAL: ( ? )
AGENT: (a Man who is (the 'Person' in story-x))
INSTRUMENT: (a .357 magnum pistol)
```

```
OBJECT: (a sandwich)
CONSEQUENCE: (the sandwich has a bullet hole)
SPECIALIZATION: (SPORT)
```

FIGURE 7.5

In lines 41 to 46 the Subject is giving a retrospective account of processes that occured automatically and which resulted in the hypotheses inhering in line 39, and represented in the 'image' which the Subject is discussing.

Line 41 of the protocol suggests that the first process was simply the triggering by elements of the problem statement of the Shooting schema, with the values of the AGENT: and INSTRUMENT: slots only loosely specified.

Lines 42 and 43 suggest that a question arose ("At what would the gun be pointing?") and line 44 indicates that an answer (the sandwich) was computed.

I argue that the remaining hypotheses from the Recursion schema are responsible for the statements. The hypotheses were these:

HYPOTHESIS-3: some action will be performed on (SANDWICH PLATE
NEWSPAPER BOOK TABLE FLOOR)

HYPOTHESIS-4: the action to be performed will be achieved first on the object at the head of the list (= SANDWICH) then on all of the other objects in the tail of the list.

With these specifications the Shooting schema would try to select a Track whose specifications matched the data. A couple of the Shooting Tracks are given below (Figure 7.6).

```
$SHOOTING (Sport)
```

```
AGENT: (a Person {DEFAULT (a Man)})
OBJECT: (an Object {DEFAULT (a Target)})
INSTRUMENT: (a gun)
GOAL: (achieve Esteem)
.....

$SHOOTING (Hunting)
AGENT: (a Person {DEFAULT (a Man)})
OBJECT: (an Animate-Object {DEFAULT (an Animal)})
INSTRUMENT: (a gun)
.....
```

FIGURE 7.6

The Hunting Track requires an Animate-Object, and the Sport Track an Inanimate-Object in their respective OBJECT: slots. The Shooting schema tries to fit 'SANDWICH' (the expectation handed to it from the hypotheses from the Recursion schema), to the various Tracks and instantiates the Sport schema as a result of its restriction on the OBJECT: slot, which is that it requires an Inanimate object. (The presumption is that the Shooting schema can acquire this information, although how is not here specified.)

In line 45 the Subject further specifies the image referred to in line 39. The Subject had imagined the bullet going into the sandwich, the plate, and the table. The list does not include 'NEWSPAPER' and 'BOOK', as required by the last hypothesis hypothesis from Recursion, but the direction of movement is as predicted (the person could have shot 'at' the sandwich, or the book, without having tried to shoot all of the objects). The interesting factor here is that after mentioning 'TABLE' the subject hesitates and then continues, saying (line 46) I don't know how far the bullets go...." This suggests that general knowledge of the constitution of objects and what effect it has on the paths of bullets has been used, and a possible conflict with Recursion's fourth hypothesis recorded: the bullet might not get through the table, much less the floor. Thus, Recursion's expectation that all the objects in its APPLIES-TO: list would achieve the effect 'HAS BULLETHOLE' would be contradicted. On the other hand, if the gun was powerful enough, the bullet would penetrate it. The result is that a search for information about the power of a '.357 magnum pistol' should be initiated. That such a search was active is reflected in S8's comment at line 54 of the protocol.

```
RECURSIVE-PROCEDURE
GOAL: (ForEvery x In (my applies-to) do
          (achieve (my effect) through (my action) x))
ACTION: (a SHOOTUP)
EFFECT: (Add-Pattern (HAS BULLETHOLE) ForEvery x In (my applies-to))
APPLIES-TO: (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR)
LOOP-FOR: (ForEvery x In (my applies-to))
```

A Plan constructed from the indicated slots, and deposited in Working Memory, is this:

```
  ((procedure-name: shootup)
   (plan: (write procedure name)
        (add-pattern parameter has bullethole)  <= from ACTION: slot
        ((for-the-chain parameter on var2)      <= from GOAL: slot
         (reapply var2 has bullethole)))
   (tr-list: (on (sandwich plate newspaper book table floor)))
   (required-result: (sandwich plate (has bullethole)))
```

This plan is the presumed starting point for the production system model of coding and program evaluation presented in Chapter 8.

7.4 S5'S PROTOCOL (READING PHASE)

The segment of S5's protocol, presented below, includes the Subject's comments only on the first three lines of the problem statement. This extract is sufficient to indicate S5's general approach to the problem, and the difficulty S5 had in applying programming

Knowledge to the problem understanding processes. Here is the protocol:

"On page eighty of Units three to four we looked at a method for making a particular inference 'keep on happening'."	READING
1 "On page 80...."	RE-READING
2 Um....	?
3 Well I'm very surprised, because I didn't think it was going to be detailed like this.	META-COMMENT
4 And I think....	META-COMMENT
5 "An inference 'Keep on happening'"	RE-READING
6 Um....	?
7 Well....	?
8 "'Keep on happening'"	RE-READING
9 Um....	?
10 That reminds me of something.	META-COMMENT
11 I'm trying to remember what the words were that it reminds me of.	META-COMMENT
12 Um....	?
13 "'Keep on happening'"	RE-READING
14 Um....	?
15 FOR EACH CASE OF, it reminds me of FOR EACH CASE OF.	S-A(C)
16 And then INFERENCE is where you have, um....	S-A(C)
17 Two statements, and the database is able to form, um....	S-A(A)/PROBE
18 Or the PROGRAM is able to modify the DATABASE by, um....	S-A(A)/PROBE
19 Putting in a new relationship on the basis of relating to relationships that were already in the DATABASE.	S-A(A)
20 So you could have two TRIPLES, um, and they would imply, or the....	S-A(A)/ ELABORATION
21 The PROGRAM has been made so that, um....	S-A(A)

22 The existing two triples will form an INFERENCE which is formed....	S-A(A)/ ELABORATION
23 Which is stated on....	ELABORATION
24 The DATABASE as a third TRIPLE.	S-A(A)
25 That's what an INFERENCE is.	S-A(A)
26 "We looked at a method for making an inference...."	RE-READING
27 I can't remember what it is....	META-COMMENT
28 Is that enough?	QUERY
E: Whenever you're ready to go on to sentence two.	EXP-COMMENT
"In this option you're asked to imagine a state of the world in which there are six objects...."	READING
29 It's not about the computer at all!	HYPOTHESIS
"A sandwich, a plate, a newspaper,"	READING
30 That's three.	ELABORATION
31 "Sandwich, plate, newspaper"	RE-READING
"A book, a table, and a floor."	READING
32 That's just like a DATABASE, isn't it?	S-A(C)
33 A very limited world.	META-COMMENT
"Further...."	READING
34 Oh, you want one sentence at a time.	QUERY
35 "In this option, in this option...."	RE-READING
36 Um....	?
37 Well, there must be going to be other options, and this must be the first one.	HYPOTHESIS
38 And these are the conditions which apply for this first option.	HYPOTHESIS/ ELABORATION
39 A limited world with six things in it.	META-COMMENT
40 Um....	?
41 And they might be on top of each other or beside each other.	HYPOTHESIS

42 Um.... ?

43 "In this option you are asked to imagine a state of the world...." RE-READING

44 Um.... ?

"Further, this hypothetical world is highly structured...." READING

45 Okay. ?

"....the sandwich is lying in the center of the plate, which is sitting on the newspaper, which is lying on the book which is on the table, which of course rests on the floor." READING

46 So the FLOOR's on the bottom. ELABORATION

47 And everything's piled up.... ELABORATION

48 In the normal way, with the SANDWICH on the top....
HYPOTHESIS/S-A(C)

49 And.... ?

50 Well that sentence is simply stating, um.... META-COMMENT

51 The contents of the problem, or the content of a proposition that could be considered..... META-COMMENT

52 The FLOOR looks very odd, doesn't it? ELABORATION

53 I keep looking at FLOOR.... RE-READING/

META-COMMENT

54 It begins to look peculiar, um.... META-COMMENT

7.4.1 Overview Of The Protocol.

The recursion schema was not triggered anywhere in the reading phase of this Subject. In Chapter 8, when the post-reading phase of this Subject's behaviour is examined, it will be seen that recursion is a concept about which S5 had some odd notions. At the end of the reading phase S5 was not at all sure of what was required of her, whether she was meant actually to create a program, and if so, how to go about it. But this is a problem for discussion in the next Chapter of the thesis.

7.4.2 The First Problem Statement.

The concepts with which S5 attempted to deal after reading this line of the Problem Statement were: FOR EACH CASE OF and 'inference'. Most of the response to the first line had to do with associating the label 'Keep on happening' with some concept in memory, which association was finally achieved on line 15 of the protocol. In the rest of the comments on this first line, the Subject is seen PROBEing her knowledge of inferencing programs, what they do, and how they do it.

7.4.3 The Second Problem Statement.

The first HYPOTHESIS offered by S5 - which is incorrect, of course - occurs after reading this line. The objects mentioned in the line, and presumably the relations between them, triggers DATABASE. The Subject's second HYPOTHESIS is not strictly relevant to the problem in hand, but the Subject was not to know this. The third HYPOTHESIS is that the objects may have one relationship with one another, or a different relationship.

7.4.4 The Third Problem Statement.

The HYPOTHESIS stated here is that the 'sandwich is on the top' of the pile of objects which are 'stacked up in the normal way'. It is at first not an easy matter to understand what is meant by this last comment, since normally sandwiches on plates are not stacked on newspapers and books, etc. What the Subject probably meant was that there was nothing especially odd about the arrangement of objects - that the FLOOR was not on top of the TABLE, and so on. There is no way to tell what thoughts are reflected in protocol lines 52-54.

7.5 EVALUATION OF THE THEORY.

The behaviour of S8 lends some support to the theory of problem solving presented in the first section of this Chapter. In a way, the behaviour of S5 also lends a little support. Each of the protocols will be considered in turn.

7.5.1 The Pattern Of S8's Behaviour.

The schemas triggered by problem statement elements (first line) were high level programming methods: iteration and recursion. The activation of these schemas led to 1) DISCONFIRMATION of the first, and 2) HYPOTHESIS generation from the second (recursion). This HYPOTHESIS was supported at line 3 of the Problem statement. After reading the second line, the pattern was 1) schema activation, followed by 2) HYPOTHESIS generation. HYPOTHESIS generation is the main feature of the

Subject's response to lines 3, 5, and 6. After reading lines 7 and 8 the Subject's comments indicate CONFIRMATION of earlier HYPOTHESES. By the time the Subject comes to line 9 of the Problem Statement, it is possible to design a program 'in the head', without recourse to consulting the Programming Manual, or making notes.

7.5.2 The Pattern Of S5's Behaviour.

It is only in the 14th line of the protocol that schema activation is scored for S5's protocol. The associative activation of several other concepts occurs subsequent to the cued activation of 'Inference' but nothing occurs subsequent to the activation of 'FOR EACH CASE OF'. There are no instances of HYPOTHESIS generation subsequent to the occurences of schema activation in the comments on this line of the Problem Statement. The first HYPOTHESIS to occur is in response to reading the second line of the Problem Statement - an incorrect hypothesis, as can be seen. Towards the end of the Subject's comments on this line of the Problem Statement an HYPOTHESIS concerning the layout of the objects is offered, but there is no apparent support for the theory in the form of the hypothesis. Finally, after reading the third line of the Problem Statement the Subject is seen to be concerned with the layout of the objects, but there is no evidence in the preceding lines that programming knowledge had a role to play in directing attention as indicated in the theory.

In sum, there is both supporting and disconfirming evidence on the 'interactionist' theory presented above. There is much in the comments of S5 which suggest that this Subject has some difficulty retrieving concepts, and also in accessing the knowledge associated with the concepts eventually retrieved. This is reflected in the amount of RE-READING that occurs in response to the first three lines of the Problem Statement, and in the number of PROBEs initiated. The safest statement to be made at the moment would seem to be that the interactionist theory perhaps only holds in circumstances where novices have fairly well elaborated programming knowledge, that is, when several basic concepts have been well integrated by a model of a process, such as the Loop model of recursion. Without such integration, it seems from S5's behaviour that real world knowledge and programming knowledge are two more or less separate 'packages' which are picked up and worked on in turn.

CHAPTER 8

PRODUCTION SYSTEM MODELS OF CODING AND PROGRAM EVALUATION

8.1 INTRODUCTION

A major goal motivating the research reported in this thesis is to provide computer models of the processes novices use in writing computer programs. Although this goal has not been 100% achieved, some aspects of the behaviour have been implemented in computer programs. In this chapter, two production systems (PS) will be described. The first PS is a model of the Coding and Program Evaluation phase of Subject S8's programming behaviour. S8's performance while coding the 'BULLETHOLE' problem will be described in some detail in the next section, and then the PS model will be presented.

As indicated in Chapter 4, it is likely that many novices have no very extensive knowledge of recursion. However, it was also suggested that novices with a Syntactic, or Magic model of recursion would be able to recognize recursive procedures having a particular surface structure, and be able to successfully predict whether such a program would have a required effect on databases of a certain form. It was presumed that imitation of programs constitutes a fairly low level of problem solving and would cause the person using such a strategy little difficulty. In Section 7.4 there will be a discussion of this misguided belief with respect to the many solution attempts of one of the Subjects (S5) who attempted to write a program through imitation of the 'INFECT' program on page 80 of the Programming Manual. An 'imitation' PS customized to model one aspect of the imitation strategy of S5 will be described in the last section of this chapter.

8.2 THE 'BULLETHOLE' PROBLEM: S8'S PROTOCOL (CODING & EVALUATION PHASE)

The protocol indicated that S8 went through two stages of programming activity once the problem had been understood. The first of these was a coding stage, and the second a program evaluation stage. The two stages are indicated in the extracts from the protocol below. [The extract has been slightly edited. There was a misprint in the Problem Statement - 'SHOOTUP' had been printed as 'SOOTUP' - and this was noticed by the Subject, and commented upon. The misprint had no effects of confusion, so they and all inessential details have been deleted. The protocol lines that remain are quoted verbatim, and occured in the order in which they are presented.]

8.2.1 The Coding Stage.

In the first stage, the program was typed in at the terminal, with glances back at the problem statement to gather information on the name of the required procedure (first few lines of the extract), to check the name of the relation linking nodes, and so forth. The entire program is written as though the Subject has a well worked out mental plan for the program, but has to set up a search in the problem statement for the plans variables. Here is the extract from the protocol in which the coding of the program occurs.

> So, it's TO SHOOTUP (starts at the terminal)

E: Do you want to write it down?

> Ummm....

E: I mean, go ahead and write it down....

> No, I was just looking at what to call it. Is it okay to just do it on....

E: Yeah, yeah.

Types TO SHOOTUP /X/

> So we want that to have HAS BULLETHOLE.

Types NOTE

> Uh, NOTE....

E: Are you going to have problems there?

> Yes, I haven't put a line.

Deletes NOTE. Types 1 NOTE /X/ HAS BULLETHOLE

> Now I just do

Types RETURN

> ....that.

> Two....

Types 2 CHECK

> CHECK....CHECK X

Types /X/

Looks at problem statement.

> Just ON, QUESTION MARK

Types ON, ?, RETURN

Display shows: If Present:

I'll have to put a function....type SHOOTUP.

Ummm.... What's the situation now?

IF PRESENT:

Types SHOOTUP

....SHOOTUP ASTERISK

Types *

Yeah?

Ummm, EXIT

Types ;, EXIT, RETURN

Display shows If Absent:

Is that right?

Types EXIT

IF ABSENT EXIT

Uh, will that work? (Begins mumbling-reading what has been typed.)

8.2.2 The Evaluation stage.

Once the program had been typed in but before typing 'DONE' (indicating the end of the program definition) the Subject went into the second stage, which involved evaluating the program by mentally instantiating variable values, and simulating the behaviour of the program. For example, the Subject begins by naming the procedure, then the procedure plus parameter, and then mentally makes the parameter equal to the first node the procedure is to operate on: 'SANDWICH'.

Uh, I was wondering if....I thought I should check it at this stage....

E: Yeah. Are you checking it? I mean that's what I want to know....

Yeah, I'm thinking about how it goes through....

TO SHOOTUP, SHOOTUP X, let's say X is a SANDWICH....

E: Right.

First of all it NOTEs in the database....X HAS BULLETHOLE

It then CHECKs whether X is ON anything....

If it is, if it's PRESENT it does SHOOTUP to the next thing like....

Looks at problem statement

X is ON PLATE so it will do that to PLATE....

So that should keep doing that, PLATEs on, CHECK, something, so on and so on....

If it's not on anything it's okay to just EXIT, isn't it?

E: Right.

So I write RETURN, DONE.

Types RETURN DONE RETURN

Display shows OK I NOW KNOW HOW TO 'SHOOTUP' /X/

8.3 THE PRODUCTION SYSTEM MODEL OF S8'S PROGRAMMING BEHAVIOUR.

In this section the annotated production system model of the coding and evaluation process is presented.

The production systems in this chapter were designed by myself, but were implemented by Rick Evertz, (a research assistant in our laboratory) in his own production system language 'POPSI'.

The system's working memory contains a representation of the problem which has been derived from information in the Recursion schema. The problem representation consists of a plan for coding the program, plus the expected result of running the program. The plan would have been constructed by combining the information gathered during the reading phase with the information in the SURFACE-STRUCTURE: slot of the schema. These interactions have not been modelled. This is the way the problem is represented in working memory at the beginning of the simulation (this is the 'plan' discussed in Chapter 7):

```
(defv working-memory
  ((procedure-name: shootup)
   (plan: (write procedure name)
        (add-pattern parameter has bullethole)  <= from ACTION: slot
        ((for-the-chain parameter on var2)      <= from GOAL: slot
         (reapply var2 has bullethole)))
   (tr-list: (on (sandwich plate newspaper book table floor)))
   (required-result: (sandwich plate (has bullethole)))
   ([em] SOLO:)
   ))
```

These are the production rules:

```
;*
;*  The rule-set consists of two packets: <CODING-PKT> and
;*  <EVALUATION-PKT>.

;****************     THIS IS THE START OF THE CODING PACKET
 ************;


; If you are satisfied that the program will work,
; then write 'DONE' to external memory, and halt...

    (CM1     ((program t))
          ==>
          ((*em* DONE)
           (*halt*)))


; If the next part of the plan is to write the procedure name,
; and you know what the procedure name is,
; then deposit the rest of the plan in working memory,
; and type 'TO procedure-name /X/'...


    (C2a     ((plan: (write procedure name) &rest)
           (procedure-name: =procname))
          ==>
          ((plan: &rest)
           (*em* TO  =procname  x)))
```

```
; If the next part of the plan is to
; add the pattern 'something relation something-else',
; and SOLO has just prompted you with a line-number,
; then deposit the rest of the plan,
; and type 'NOTE something relation something-else...

   (C2b    ((plan: (add-pattern parameter =rel =node2) &rest)
            ([em] (*linenumberp* (=n)) & (TO = =var) &))
           ==>
           ((plan: &rest)
            (*em* NOTE  =var  =rel  =node2)))


; If the next part of the plan is that 'for-the-chain' thing,
; and SOLO prompts with a line-number,
; then deposit the rest of the plan in working memory,
; and type 'CHECK some-parameter some-relation question-mark'...

   (C2c     ((plan: ((for-the-chain parameter =rel var2) (reapply var2
 = =))
                     &rest)
             ([em] (*linenumberp* (=n)) & (TO = =var) &))
            ==>
            ((plan: &rest)
             (*em* CHECK  =var  =rel  ?)))


; If SOLO has just prompted you with 'If Present:' and the
; next part of the plan is to reapply the procedure,
; then type the 'procedure name', a 'star' and 'EXIT'.
; N.B. the procedure name is obtained from the 'TO ...' line
; of external memory.

   (C3a    (([em] (= A If Present:) & (TO =procname =) &)
           (plan: (= (reapply = = =))))
           ==>
           ((*em* =procname  *  :  EXIT)))


; When SOLO prompts you with 'If Absent:', if the plan is
; null then just type 'EXIT' and invoke the evaluation packet.

   (C3b    (([em] (= B If Absent:) &)
           (plan:))
           ==>
           ((*em* EXIT)
            (<evaluation-pKt>)))


;***************    THIS IS THE START OF THE EVALUATION PACKET   ***

  <evaluation-pKt>


; At the start of evaluation (i.e. before a mental program pointer has
; been generated), attend to the top line of the procedure, and mentally
; note that at this point, the relevant subset of the database is empty.

   (EV0   ((*not* (program-pointer: &))
          ([em] & (TO =procname =var) &))
         ==>
         ((program-pointer: (TO =procname =var))
          (database: (())))))


; If you are looking at the top line of the procedure, and this contains
; a formal parameter, then mentally note that you will next be looking
 at the
; next line, and deposit a subgoal to ascertain the value of that
 parameter.
; (alternatively one could increment the program pointer after
 ascertaining
;  the value of the formal parameter)

   (EV1   ((program-pointer: (TO = =var))
          ([em] & =nextline =nextlinenum (TO = =var) &))
         ==>
         ((*subst* (program-pointer: =nextline =nextlinenum)
                   for
                   (program-pointer: &))
          (ascertain =var)))


; If the current line is a 'NOTE' and you know what the node
; you are noting is then add its current binding to your little
; mental database.

   (EV2   ((program-pointer: (NOTE =node1 =rel =node2) =)
          (=node1 is =currentbinding)
          (database: (&nodes =prop))
          ([em] & =nextline =nextlinenum (NOTE &) &))
         ==>
         ((*subst* (program-pointer: =nextline =nextlinenum)
                   for
                   (program-pointer: &))
          (*subst* (database: (&nodes =currentbinding (=rel =node2)))
                   for
                   (database: (&nodes =prop)))))


; If the next line is a 'CHECK' then get what fills the '?' slot off
; of the tr-list, and say what it is.

   (EV3   ((program-pointer: (CHECK =node1 =rel ?) =)
          (=node1 is =currentbinding)
          (tr-list: (=rel (& =currentbinding =nextnode &)))
          ([em] & =nextline =nextlinenum (CHECK &) &))
         ==>
         ((*say* (=node1 =rel =nextnode))
          (*subst* (program-pointer: =nextline =nextlinenum)
                   for
                   (program-pointer: &))))
```

```
; If you are looking at the line that does the recursive call,
; then increment the program pointer, and deposit a new pointer,
; which points to the top line of the procedure.  So we now have
; two pointers: the old one which the PS would come back to if it
; had to after sorting out what happens in the recursive call, and
; a new one which has precedence because it was deposited more
; recently (i.e. is a more recent subgoal).

    (EV4   ((program-pointer: (=procname * : =) (= A If Present:))
            (=var =rel =node2)
            ([em] & =nextline
                   =nextlinenum
                   (=procname * : =)
                 & (TO =procname =) &))
           ==>
           ((*subst* (program-pointer: =nextline =nextlinenum)
                     for
                     (program-pointer: &))
            (program-pointer: (TO =procname =var))))


; The database is the same as the required-result, so deposit (program
 t),
; and invoke the coding-packet.

    (DONE  ((database: =currentstate)
            (required-result: =currentstate))
           ==>
           ((program t)
            ((coding-pkt))))


; If you want to ascertain the value of some parameter, then its
; value is the next node on the tr-list.  The next node on the tr-list
; is merely the node which comes after the last node which you looked
; at, in other words the one after '&nodes'.
; N.B. at the very start of program evaluation, '&nodes' is null, and
; so the next node would in fact be the first node.  In other words,
; the '&matcher' in a lefthandside can match against anything, including
; nothing.

    (APR1  ((ascertain =var)
            (database: (&nodes))
            (tr-list: (= (&nodes =nextnode &))))
           ==>
           ((*say* (=var is =nextnode))))

))
```

DISCUSSION: The decision to tie the productions together in packets is justified by S8's dedicated application of first one method (coding) and then another (evaluation). However, it would be a simplification to suggest that S8 shows a single-minded attention to one phase of programming without keeping sight of other phases. Remember that the protocol shows the Subject suspending coding behaviour (the Subject does not type 'DONE' until the result from the evaluation phase is in) rather than ending it suddenly. The protocol suggests that this Subject is in a transition phase between novice and expert behaviour with respect to commenting the behaviour being produced. In the coding phase the Subject is working at a purely abstract level, not once instantiating values for the formal parameter, and so forth. This suggests that the problem variables have already been assimilated to an internalized representation of the program that is being designed 'on the run' as it were (also see the reading phase of this Subject's protocol; Chapter 7). Only during program evaluation does the Subject switch over into 'careful' mode, although even here the Subject's confidence in the algorithm is displayed. For example, although the formal parameter is instantiated with the node at the head of the Chain relation list, and the required side effect to the database carefully checked, the behaviour of the recursive segment is 'summarized' by the Subject after it has been ascertained that the next node will be generated, as required, and that the side effect to this node also will occur as required. This careful evaluation of the effects of code is a high level skill which is not available to all novices, as will be seen in the next main section, below. In the rest of this section there is presented a sample run of the PS designed to model S8's coding and evaluation behaviour. The trace very closely matches the behaviour exhibited in the protocol, and as such represents a clear reflection of the processes - at a general level of anlaysis - that have occured.

```
; HERE IS A SAMPLE RUN OF THE ABOVE PRODUCTION SYSTEM:

Working memory is :

((PLAN:)
 (PLAN: ((FOR-THE-CHAIN PARAMETER ON VAR2)
          (REAPPLY VAR2 HAS BULLETHOLE)))
 (PLAN: (ADD-PATTERN PARAMETER HAS BULLETHOLE)
         ((FOR-THE-CHAIN PARAMETER ON VAR2)
          (REAPPLY VAR2 HAS BULLETHOLE)))
 (PROCEDURE-NAME: SHOOTUP)
 (PLAN: (WRITE PROCEDURE NAME)
         (ADD-PATTERN PARAMETER HAS BULLETHOLE)
         ((FOR-THE-CHAIN PARAMETER ON VAR2)
          (REAPPLY VAR2 HAS BULLETHOLE)))
 (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR)))
 (REQUIRED-RESULT: (SANDWICH PLATE (HAS BULLETHOLE))))

External memory is :

([EM] (20 B IF ABSENT:)
      (SHOOTUP * : EXIT)
      (20 A IF PRESENT:)
      (CHECK x ON ?)
      (20)
      (NOTE x HAS BULLETHOLE)
      (10)
      (TO SHOOTUP x)
      SOLO:)

Firing rule: C3B

C3B
(([EM] (= B IF ABSENT:) &) (PLAN:))
==>
((*EM* EXIT) (<EVALUATION-PKT>))

{Chosen on the basis of RECENCY}


External memory has had: (EXIT) added.

....30:


External memory has had: (30) added by solo.

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Working memory is :

((PLAN:)
 (PLAN: ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PLAN: (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PROCEDURE-NAME: SHOOTUP)
 (PLAN: (WRITE PROCEDURE NAME)
        (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR)))
 (REQUIRED-RESULT: (SANDWICH PLATE (HAS BULLETHOLE))))

External memory is :

([EM] (30)
      (EXIT)
      (20 B IF ABSENT:)
      (SHOOTUP * : EXIT)
      (20 A IF PRESENT:)
      (CHECK x ON ?)
      (20)
      (NOTE x HAS BULLETHOLE)
      (10)
      (TO SHOOTUP x)
      SOLO:)

Firing rule: EV0

EV0
((*NOT* (PROGRAM-POINTER: &)) ([EM] & (TO =PROCNAME =vAR) &))
==>
((PROGRAM-POINTER: (TO =PROCNAME =vAR)) (DATABASE: (NIL)))
```

```
{Chosen on the basis of RECENCY}

Deposited:  (PROGRAM-POINTER: (TO SHOOTUP x))

Deposited:  (DATABASE: (NIL))

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Working memory is :

((DATABASE: (NIL))
 (PROGRAM-POINTER: (TO SHOOTUP x))
 (PLAN:)
 (PLAN: ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PLAN: (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PROCEDURE-NAME: SHOOTUP)
 (PLAN: (WRITE PROCEDURE NAME)
        (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR)))
 (REQUIRED-RESULT: (SANDWICH PLATE (HAS BULLETHOLE))))

External memory is :

([EM] (30)
      (EXIT)
      (20 B IF ABSENT:)
      (SHOOTUP * : EXIT)
      (20 A IF PRESENT:)
      (CHECK x ON ?)
      (20)
      (NOTE x HAS BULLETHOLE)
      (10)
      (TO SHOOTUP x)
      SOLO:)

Firing rule: EV1

EV1
((PROGRAM-POINTER: (TO = =vAR))
 ([EM] & =NEXTLINE =NEXTLINENUM (TO = =vAR) &))
==>
((*SUBST* (PROGRAM-POINTER: =NEXTLINE =NEXTLINENUM)
          FOR
          (PROGRAM-POINTER: (TO = =vAR)))
 (ASCERTAIN =vAR))

{Chosen on the basis of RECENCY}

Deleted:  (PROGRAM-POINTER: (TO = x))
```

```
Deposited:  (PROGRAM-POINTER: (NOTE x HAS BULLETHOLE) (10))

Deposited:  (ASCERTAIN x)

←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←

Working memory is :

((ASCERTAIN x)
 (PROGRAM-POINTER: (NOTE x HAS BULLETHOLE) (10))
 (DATABASE: (NIL))
 (PLAN:)
 (PLAN: ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PLAN: (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PROCEDURE-NAME: SHOOTUP)
 (PLAN: (WRITE PROCEDURE NAME)
        (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR)))
 (REQUIRED-RESULT: (SANDWICH PLATE (HAS BULLETHOLE))))

External memory is :

([EM] (30)
      (EXIT)
      (20 B IF ABSENT:)
      (SHOOTUP * : EXIT)
      (20 A IF PRESENT:)
      (CHECK x ON ?)
      (20)
      (NOTE x HAS BULLETHOLE)
      (10)
      (TO SHOOTUP x)
      SOLO:)

Firing rule: APR1

APR1
((ASCERTAIN =VAR)
 (DATABASE: (NIL))
 (TR-LIST: (=REL (=FIRSTNODE &REST))))
==>
((*SUBST* (TR-LIST: (=REL (=FIRSTNODE &REST))
                    (LOOKING AT: =FIRSTNODE))
          FOR
          (TR-LIST: (=REL (=FIRSTNODE &REST))))
 (*SAY* (=VAR IS =FIRSTNODE)))

{Chosen on the basis of RECENCY}

Deleted:  (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR)))
```

```
Deposited:  (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR))
                      (LOOKING AT: SANDWICH))

 "(x IS SANDWICH)"


Deposited:  (x IS SANDWICH)

←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←

Working memory is :

((x IS SANDWICH)
 (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR))
           (LOOKING AT: SANDWICH))
 (ASCERTAIN x)
 (PROGRAM-POINTER: (NOTE x HAS BULLETHOLE) (10))
 (DATABASE: (NIL))
 (PLAN:)
 (PLAN: ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PLAN: (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PROCEDURE-NAME: SHOOTUP)
 (PLAN: (WRITE PROCEDURE NAME)
        (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (REQUIRED-RESULT: (SANDWICH PLATE (HAS BULLETHOLE))))

External memory is :

([EM] (30)
      (EXIT)
      (20 B IF ABSENT:)
      (SHOOTUP * : EXIT)
      (20 A IF PRESENT:)
      (CHECK x ON ?)
      (20)
      (NOTE x HAS BULLETHOLE)
      (10)
      (TO SHOOTUP x)
      SOLO:)

Firing rule: EV2

EV2
((PROGRAM-POINTER: (NOTE =NODE1 =REL =NODE2) =)
 (=NODE1 IS =CURRENTBINDING)
 (DATABASE: (&NODES =PROP))
 (TR-LIST: (=RELATION =LIST) =LOOKINGAT)
 ([EM] & =NEXTLINE =NEXTLINENUM (NOTE &) &))
==>
((*SUBST* (TR-LIST: (=RELATION =LIST) (LOOKING AT: =CURRENTBINDING))
          FOR
          (TR-LIST: (=RELATION =LIST) =LOOKINGAT))
 (*SUBST* (DATABASE: (&NODES =CURRENTBINDING (=REL =NODE2)))
          FOR
          (DATABASE: (&NODES =PROP)))
 (*SUBST* (PROGRAM-POINTER: =NEXTLINE =NEXTLINENUM)
          FOR
          (PROGRAM-POINTER: (NOTE =NODE1 =REL =NODE2) =)))

{Chosen on the basis of RECENCY}

Deleted:  (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR))
                    (LOOKING AT: SANDWICH))

Deposited:  (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR))
                      (LOOKING AT: SANDWICH))

Deleted:  (DATABASE: (NIL))

Deposited:  (DATABASE: (SANDWICH (HAS BULLETHOLE)))

Deleted:  (PROGRAM-POINTER: (NOTE x HAS BULLETHOLE) =)

Deposited:  (PROGRAM-POINTER: (CHECK x ON ?) (20))

←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←

Working memory is :

((PROGRAM-POINTER: (CHECK x ON ?) (20))
 (DATABASE: (SANDWICH (HAS BULLETHOLE)))
 (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR))
           (LOOKING AT: SANDWICH))
 (x IS SANDWICH)
 (ASCERTAIN x)
 (PLAN:)
 (PLAN: ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PLAN: (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PROCEDURE-NAME: SHOOTUP)
 (PLAN: (WRITE PROCEDURE NAME)
        (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (REQUIRED-RESULT: (SANDWICH PLATE (HAS BULLETHOLE))))

External memory is :

([EM] (30)
      (EXIT)
      (20 B IF ABSENT:)
      (SHOOTUP * : EXIT)
      (20 A IF PRESENT:)
      (CHECK x ON ?)
      (20)
      (NOTE x HAS BULLETHOLE)
      (10)
      (TO SHOOTUP x)
      SOLO:)

Firing rule: EV3

EV3
((PROGRAM-POINTER: (CHECK =NODE1 =REL ?) =CURRENTLINENUM)
 (=NODE1 IS =CURRENTBINDING)
 (TR-LIST: (=REL (& =CURRENTBINDING =NEXTNODE &REST))
          (LOOKING AT: =CURRENTBINDING))
 ([EM] & =NEXTLINE =NEXTLINENUM (CHECK &) &))
==>
((*SAY* (=NODE1 =REL =NEXTNODE))
 (*SUBST* (PROGRAM-POINTER: =NEXTLINE =NEXTLINENUM)
          FOR
          (PROGRAM-POINTER: (CHECK =NODE1 =REL ?) =CURRENTLINENUM)))

{Chosen on the basis of RECENCY}

  "(x ON PLATE)"

Deposited:  (x ON PLATE)

Deleted:  (PROGRAM-POINTER: (CHECK x ON ?) (20))

Deposited:  (PROGRAM-POINTER: (SHOOTUP * : EXIT) (20 A IF PRESENT:))$q$q

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Working memory is :

((PROGRAM-POINTER: (SHOOTUP * : EXIT) (20 A IF PRESENT:))
 (x ON PLATE)
 (DATABASE: (SANDWICH (HAS BULLETHOLE)))
 (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR))
           (LOOKING AT: SANDWICH))
 (x IS SANDWICH)
 (ASCERTAIN x)
 (PLAN:)
 (PLAN: ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PLAN: (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PROCEDURE-NAME: SHOOTUP)
 (PLAN: (WRITE PROCEDURE NAME)
        (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (REQUIRED-RESULT: (SANDWICH PLATE (HAS BULLETHOLE))))

External memory is :

([EM] (30)
      (EXIT)
      (20 B IF ABSENT:)
      (SHOOTUP * : EXIT)
      (20 A IF PRESENT:)
      (CHECK x ON ?)
      (20)
      (NOTE x HAS BULLETHOLE)
      (10)
      (TO SHOOTUP x)
      SOLO:)

Firing rule: EV4

EV4
((PROGRAM-POINTER: (=PROCNAME * : =) (= A IF PRESENT:))
 (=vAR =REL =NODE2)
 ([EM] &
       =NEXTLINE
       =NEXTLINENUM
       (=PROCNAME * : =)
       &
       (TO =PROCNAME =)
       &))
==>
((*SUBST* (PROGRAM-POINTER: =NEXTLINE =NEXTLINENUM)
          FOR
          (PROGRAM-POINTER: (=PROCNAME * : =) (= A IF PRESENT:)))
 (PROGRAM-POINTER: (TO =PROCNAME =vAR)))

{Chosen on the basis of RECENCY}

Deleted:  (PROGRAM-POINTER: (SHOOTUP * : =) (= A IF PRESENT:))

Deposited:  (PROGRAM-POINTER: (EXIT) (20 B IF ABSENT:))

Deposited:  (PROGRAM-POINTER: (TO SHOOTUP x))$q$q

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Working memory is :

((PROGRAM-POINTER: (TO SHOOTUP x))
 (PROGRAM-POINTER: (EXIT) (20 B IF ABSENT:))
 (x ON PLATE)
 (DATABASE: (SANDWICH (HAS BULLETHOLE)))
 (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR))
           (LOOKING AT: SANDWICH))
 (x IS SANDWICH)
 (ASCERTAIN x)
 (PLAN:)
 (PLAN: ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PLAN: (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PROCEDURE-NAME: SHOOTUP)
 (PLAN: (WRITE PROCEDURE NAME)
        (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (REQUIRED-RESULT: (SANDWICH PLATE (HAS BULLETHOLE))))

External memory is :

([EM] (30)
      (EXIT)
      (20 B IF ABSENT:)
      (SHOOTUP * : EXIT)
      (20 A IF PRESENT:)
      (CHECK x ON ?)
      (20)
      (NOTE x HAS BULLETHOLE)
      (10)
      (TO SHOOTUP x)
      SOLO:)

Firing rule: EV1

EV1
((PROGRAM-POINTER: (TO = =vAR))
 ([EM] & =NEXTLINE =NEXTLINENUM (TO = =vAR) &))
==>
((*SUBST* (PROGRAM-POINTER: =NEXTLINE =NEXTLINENUM)
          FOR
          (PROGRAM-POINTER: (TO = =vAR)))
 (ASCERTAIN =vAR))

{Chosen on the basis of RECENCY}

Deleted:  (PROGRAM-POINTER: (TO = x))

Deposited:  (PROGRAM-POINTER: (NOTE x HAS BULLETHOLE) (10))

Deposited:  (ASCERTAIN x)$q$q

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Working memory is :

((ASCERTAIN x)
 (PROGRAM-POINTER: (NOTE x HAS BULLETHOLE) (10))
 (PROGRAM-POINTER: (EXIT) (20 B IF ABSENT:))
 (x ON PLATE)
 (DATABASE: (SANDWICH (HAS BULLETHOLE)))
 (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR))
           (LOOKING AT: SANDWICH))
 (x IS SANDWICH)
 (ASCERTAIN x)
 (PLAN:)
 (PLAN: ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PLAN: (ADD-PATTERN PARAMETER HAS BULLETHOLE)
       ((FOR-THE-CHAIN PARAMETER ON VAR2)
        (REAPPLY VAR2 HAS BULLETHOLE)))
 (PROCEDURE-NAME: SHOOTUP)
 (PLAN: (WRITE PROCEDURE NAME)
        (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (REQUIRED-RESULT: (SANDWICH PLATE (HAS BULLETHOLE))))

External memory is :

([EM] (30)
      (EXIT)
      (20 B IF ABSENT:)
      (SHOOTUP * : EXIT)
      (20 A IF PRESENT:)
      (CHECK x ON ?)
      (20)
      (NOTE x HAS BULLETHOLE)
      (10)
      (TO SHOOTUP x)
      SOLO:)

Firing rule: APR2

APR2
((ASCERTAIN =vAR)
 (DATABASE: (&NODES =CURRENTNODE =))
 (TR-LIST: (= (&NODES =CURRENTNODE =NEXTNODE &))
           (LOOKING AT: =CURRENTNODE)))
==>
((*SAY* (=vAR IS =NEXTNODE)))


{Chosen on the basis of RECENCY}

  "(x IS PLATE)"

Deposited:  (x IS PLATE)

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Working memory is :

((x IS PLATE)
 (ASCERTAIN x)
 (PROGRAM-POINTER: (NOTE x HAS BULLETHOLE) (10))
 (PROGRAM-POINTER: (EXIT) (20 B IF ABSENT:))
 (x ON PLATE)
 (DATABASE: (SANDWICH (HAS BULLETHOLE)))
 (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR))
           (LOOKING AT: SANDWICH))
 (x IS SANDWICH)
 (ASCERTAIN x)
 (PLAN:)
 (PLAN: ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PLAN: (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PROCEDURE-NAME: SHOOTUP)
 (PLAN: (WRITE PROCEDURE NAME)
        (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (REQUIRED-RESULT: (SANDWICH PLATE (HAS BULLETHOLE))))
```

External memory is :

```
([EM] (30)
      (EXIT)
      (20 B IF ABSENT:)
      (SHOOTUP * : EXIT)
      (20 A IF PRESENT:)
      (CHECK x ON ?)
      (20)
      (NOTE x HAS BULLETHOLE)
      (10)
      (TO SHOOTUP x)
      SOLO:)
```

Firing rule: EV2

```
EV2
((PROGRAM-POINTER: (NOTE =NODE1 =REL =NODE2) =)
 (=NODE1 IS =CURRENTBINDING)
 (DATABASE: (&NODES =PROP))
 (TR-LIST: (=RELATION =LIST) =LOOKINGAT)
 ([EM] & =NEXTLINE =NEXTLINENUM (NOTE &) &))
==>
((*SUBST* (TR-LIST: (=RELATION =LIST) (LOOKING AT: =CURRENTBINDING))
          FOR
          (TR-LIST: (=RELATION =LIST) =LOOKINGAT))
 (*SUBST* (DATABASE: (&NODES =CURRENTBINDING (=REL =NODE2)))
          FOR
          (DATABASE: (&NODES =PROP)))
 (*SUBST* (PROGRAM-POINTER: =NEXTLINE =NEXTLINENUM)
          FOR
          (PROGRAM-POINTER: (NOTE =NODE1 =REL =NODE2) =)))
```

{Chosen on the basis of RECENCY}

```
Deleted:   (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR))
                     (LOOKING AT: SANDWICH))

Deposited:  (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR))
                      (LOOKING AT: PLATE))

Deleted:   (DATABASE: (SANDWICH (HAS BULLETHOLE)))

Deposited:  (DATABASE: (SANDWICH PLATE (HAS BULLETHOLE)))
```

```
Deleted:  (PROGRAM-POINTER: (NOTE x HAS BULLETHOLE) =)

Deposited:  (PROGRAM-POINTER: (CHECK x ON ?) (20))

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Working memory is :

((PROGRAM-POINTER: (CHECK x ON ?) (20))
 (DATABASE: (SANDWICH PLATE (HAS BULLETHOLE)))
 (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR))
           (LOOKING AT: PLATE))
 (x IS PLATE)
 (ASCERTAIN x)
 (PROGRAM-POINTER: (EXIT) (20 B IF ABSENT:))
 (x ON PLATE)
 (x IS SANDWICH)
 (ASCERTAIN x)
 (PLAN:)
 (PLAN: ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PLAN: (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PROCEDURE-NAME: SHOOTUP)
 (PLAN: (WRITE PROCEDURE NAME)
        (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (REQUIRED-RESULT: (SANDWICH PLATE (HAS BULLETHOLE))))

External memory is :

([EM] (30)
      (EXIT)
      (20 B IF ABSENT:)
      (SHOOTUP * : EXIT)
      (20 A IF PRESENT:)
      (CHECK x ON ?)
      (20)
      (NOTE x HAS BULLETHOLE)
      (10)
      (TO SHOOTUP x)
      SOLO:)

Firing rule: DONE

DONE
((DATABASE: =CURRENTSTATE)
 (REQUIRED-RESULT: =CURRENTSTATE)
 (PROGRAM-POINTER: &))
==>
((PROGRAM T) (<CODING-PKT>))

{Chosen on the basis of RECENCY}
```

```
Deposited:  (PROGRAM T)

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Working memory is :

((PROGRAM T)
 (PROGRAM-POINTER: (CHECK x ON ?) (20))
 (DATABASE: (SANDWICH PLATE (HAS BULLETHOLE)))
 (TR-LIST: (ON (SANDWICH PLATE NEWSPAPER BOOK TABLE FLOOR))
           (LOOKING AT: PLATE))
 (x IS PLATE)
 (ASCERTAIN x)
 (PROGRAM-POINTER: (EXIT) (20 B IF ABSENT:))
 (x ON PLATE)
 (x IS SANDWICH)
 (ASCERTAIN x)
 (PLAN:)
 (PLAN: ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PLAN: (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (PROCEDURE-NAME: SHOOTUP)
 (PLAN: (WRITE PROCEDURE NAME)
        (ADD-PATTERN PARAMETER HAS BULLETHOLE)
        ((FOR-THE-CHAIN PARAMETER ON VAR2)
         (REAPPLY VAR2 HAS BULLETHOLE)))
 (REQUIRED-RESULT: (SANDWICH PLATE (HAS BULLETHOLE))))

External memory is :

([EM] (30)
      (EXIT)
      (20 B IF ABSENT:)
      (SHOOTUP * : EXIT)
      (20 A IF PRESENT:)
      (CHECK x ON ?)
      (20)
      (NOTE x HAS BULLETHOLE)
      (10)
      (TO SHOOTUP x)
      SOLO:)

Firing rule: CM1

CM1
((PROGRAM T))
==>
((*EM* DONE) (*HALT*))

{Chosen on the basis of RECENCY}


External memory has had: (DONE) added.
SHOOTUP has been successfully defined and added to your pool of
```

procedures.

```
TO SHOOTUP /X/

10 NOTE /X/---HAS--->BULLETHOLE
20 CHECK /X/---ON--->?
  A If Present: SHOOTUP * ; EXIT
  B If Absent :  ; EXIT

SOLO: QUIT

. C

@pop

[ CPU used by PHOTO (Hrs:Mins:Secs) - 0:00:10.949 ]
[ CPU used by job (Hrs:Mins:Secs) -   0:01:52.542 ]

[Recording terminated at 12/11/82 15:29:29]
```

8.4 THE 'BULLETHOLE' PROBLEM: S5'S PROTOCOL (CODING PHASE)

S5 decided to 'have a look in the book' at the end of the reading phase, and used the information in the first line of the problem statement to find a program to imitate. The decision to imitate the 'INFECT' program was settled after the Subject had attempted to write the title line of the program.

S5's lengthy attempt at solving the 'BULLETHOLE' problem can be summarized as the attempt to fit three different associative triples - '/X/ HAS BULLETHOLE', '/X/ SHOOTS ?', and '/X/ BULLETHOLE ?' - to two program segments: a Side-effect, and a GNO ← Self segment. The first
←
protocol segment given below begins at the point at which the Subject has decided to imitate the INFECT program on page 80 of the Programming Manual. S5 had been unable to decide whether the title line should contain 'TO SHOOTUP "?"', (at (1) in the extract below) and seeing that

the example program (INFECT) contained on its title line 'TO INFECT "/X/"', the Subject decided that the formal parameter for her own procedure also should be '/X/' instead of '?', which see:

> (1) TO SHOOTUP. Oh, do I need, um.... No, I want a QUESTION-MARK, space. Um, I don't need to specify the thing, um.... Do I want a VARIABLE or do I want a QUESTION-MARK? I think I'll have, um, QUESTION-MARK A. Um.... And then.... Um.... NOTE. Oh, they've got 'X' there. ....I want to delete the QUESTION-MARK and put X. I, I thought it should be a QUESTION MARK but I've decided t looking at the example on page eighty that it's X and (2) I'm going to follow that example now, and um, I'm going to type in NOTE, um.... NOTE.... TO NOTE.... TO SHOOTUP X....

S5 immediately then decided that the best strategy would be to imitate not only the title line of the INFECT program, but also to go on and try to imitate the rest of the program as well (at (2) in the protocol segment immeditely above).

Unfortunately, on first encounter with the INFECT program (when S5 was studying the Programming Manual at home, previous to her visit to the laboratory) S5 had developed some incorrect notions about recursion. First, S5 had been puzzled about the use of a procedure which "is incorporated subsequently in the same procedure" (quote taken from S5's protocol). Worse, from the INFECT program's wild-card pattern 'CHECK /X/ KISSES ?', S5 had acquired the notion that there had to be an 'active' relationship between two nodes ('KISSES' is an 'active' relationship) before a recursive process could work. Throughout the session the Subject made references to this conception of a restriction on the use of recursion, without the experimenter understanding, for some time, what exactly her problem was. The first obscure occurence of the idea is found during S5's initial attempt to imitate the INFECT program. However, even without a bizarre model of the requirements for recursion, S5 found the search for a solution to the problem pretty heavy going.

Some extracts from the protocol, illustrating S5's problems and strategy, are provided in the rest of this section. My comments follow each extract and refer to the lines having numbers in brackets in the extracts from the protocol.

> (3) Um.... Er, space NOTE, NOTE, um.... (4) X HAS FLU, SANDWICH ON PLATE. Er, SANDWICH HAS BULLETHOLE. SANDWICH ON PLATE, SANDWICH HAS BULLETHOLE. Um.... SANDWICH. NOTE.... Oh, I know, um.... Space.... I think I'll try NOTE.... SANDWICH IS ON.... Can I have IS ON for a RELATIONship or shall I just have ON....? Will ON be enough? I'll just try ON and NOTE.... Um, still SANDWICH ON PLATE, SANDWICH ON PLATE.... I'm now going to put, um.... Er.... (5) TWO.... TWO, NOTE.... Will it wait for a minute while I think? Um.... "Can you write a procedure which makes the following inference: when you shoot an object that object has a bullethole." SHOOT.... SHOOT.... Um, SHOOT.... Um, SANDWICH.... My mind's gone blank. Um.... (6) I've got to get the SHOOT in somewhere haven't I?

At (3), the first predicate - NOTE - is selected for imitation. At (4) we see the Subject's halting attempts to imitate this first line of the INFECT procedure - NOTE /X/ HAS FLU - with something from the following selection of triples: SANDWICH ON PLATE, SANDWICH HAS BULLETHOLE, and SANDWICH IS ON - (PLATE, presumably). 'Finding' an argument to replace that of the imitated predicate obviously involves also evaluating the candidate argument, and S5 is obviously deficient in evaluation functions, i.e., the Subject collects together all possible patterns and gives no reason for selecting 'SANDWICH ON PLATE'. A plausible basis for the selection is that it is the first pattern to appear in the problem's example database.

> NOTE SANDWICH ON PLATE. Er, can I have NOTE.... Well, I need to look in here, to see what it says in here. Um.... X.... X SHOOTS SANDWICH, I'll try that. NOTE SANDWICH ON PLATE, X, er.... Space, X SHOOTS, um.... SHOOTS SANDWICH.... Um.... SHOOTS SANDWICH, NOTE X HAS FLU, right, that's Step One. Now they've got Step Two. SANDWICH ON PLATE. No, I think I'll wipe all that out. Um.... I think what I'll do is, um, make a DATABASE with all these things 'ON' each other and then I will do this little PROGRAM after that. -----

A happy result of the displacement activity - creating the database - was the development of an evaluation rule for selecting the argument of the first predicate:

> And uh, then I will do this PROCEDURE, so, um.... TO SHOOTUP, er, SANDWICH. TO SHOOTUP X. Now, I'm just going to copy this one. I'm going to reinsert, um, my terms instead of their terms. Um, I'm going to have number One. One, um,

> NOTE, um, X.... SHOOTS SANDWICH, I think. Um.... "....might look something like this.... So that any object.... When you shoot an object, has a bullethole." X SHOOTS, objects. "....puts bulletholes in objects...." X. I think I'll try that. X, um, SHOOTS SANDWICH. No, no, I'll have X HAS BULLETHOLE. Um, yes, because that's the final position, like X HAS FLU, um, HAS BULLETHOLE. HAS BULLETHOLE. It hasn't got a bullethole, has it? But anyway. X HAS.... That will be the SANDWICH, we have that for the SANDWICH. Um, then Step Two will be.... Come on, come on.

Altogether, S5 made innumerable attempts, false starts, back ups, and designed five programs in her unsuccessful attempt at solving the problem (Figure 8.1). Throughout the session S5 continually changed the CHECK statements arguments in a vain attempt to produce a configuration which would output the sequence of side-effects required by the problem. As may be seen from this sequence of programs, once a criterion for accepting the first Step had been established, the situation stabilized until the very last attempt. Altogether, S5 made three attempts to get Step 2 right (Attempts 1, 2, and 3) after which both Steps 1 and 2 became stable, and Step 2A was manipulated (attempt number 4).

```
1) TO SHOOTUP /X/
   1 NOTE /X/ HAS BULLETHOLE
   2 CHECK /X/ SHOOTS ?A
    IP: SHOOTUP *A; EXIT
    IA: EXIT
   DONE
```

```
2) TO SHOOTUP /X/
   1 NOTE /X/ HAS BULLETHOLE
   2 CHECK /X/ BULLETHOLE ?A
    IP: SHOOTUP *A; EXIT
    IA: EXIT
   DONE
```

```
3) TO SHOOTUP /X/
   1 NOTE /X/ HAS BULLETHOLE
   2 CHECK /X/ HAS BULLETHOLE
    IP: SHOOTUP *A; EXIT
    IA: EXIT
   DONE
```

```
4) TO SHOOTUP /X/
   1 NOTE /X/ HAS BULLETHOLE
   2 CHECK /X/ HAS BULLETHOLE
    IP: NOTE *A HAS BULLETHOLE; EXIT
    IA: EXIT
   DONE
```

```
5) TO SHOOTUP /X/
    1 NOTE ?A HAS BULLETHOLE
    2 NOTE *A HAS BULLETHOLE
```

FIGURE 8.1

In this extract S5 is beginning work on program design 2). 'SHOOTS' has not produced the desired results, so S5 decides to replace 'SHOOTS' with 'HAS-BULLETHOLE'. The segment begins with S5 evaluating the move (1), and the move proves to be not a very satisfactory solution from S5's point of view, either in real world terms (2), or in programming terms (3), but the relation name reflects the nature of the inference that is required (4) and so the new alternative is deemed to provide some prospect of success. Throughout, S5's voices her concern to get a verb into the relation slot between the two variables.

> Well, I don't think it makes sense but I can't think what else.... Um, the mere existence of somebody HAVING FLU is meant to.... Suggest that the person being KISSed will have FLU and perhaps we can make the same assumption here, that one thing having a BULLETHOLE will have, make the next thing have a BULLETHOLE. (2) But in fact in real life that's not true, is it? Cause, I mean, it is true, but, um, it's not a verb. SHOOTS is the verb. But.... (3) We've already IMPLIED that if one thing HAS a BULLETHOLE the other thing will have, um.... Will it accept.... Well, I can always type it in and try.... And it will correct me won't it? Um.... EDIT SHOOT UP. CHECK.... CHECK X space.... Can I put.... No....

E: What were you going to suggest?

> I was going to say HAS-BULLETHOLE in the middle. That would satisfy me more.

E: How do you mean?

> To have a verb in the middle, that says HAS-BULLETHOLE. Well, I'm not very happy with that BULLETHOLE A, there. (3) That doesn't seem to be, um, a proper RELATIONSHIP. (4) Well, it's making the assumption, well, it does say in the writing that you had to assume that if the BULLETHOLE was in the top one then it went all the way through to the bottom. But I like to have a verb in there when I'm doing it. Because that makes more sense to me. But I'm going to try this now I've got to find out if it works. [S5 types the new program in at the terminal and runs it. The program puts a bullethole in the sandwich and stops.] Go on then, do something. Why doesn't it do something?

Finally, S5 turns a critical eye on the INFECT program itself. Her own programs are too consistently wrong to be the real source of trouble. The obvious explanation is that the example in the Programming Manual has something wrong with it.

> (1) Um, I'm reading this example in the text, again, on page eighty. TO INFECT X.... NOTE X HAS FLU, um.... They seem to have the same format as me, and mine doesn't work. (2) They claim theirs does work. (3) I mean, it's this BULLETHOLE business, isn't it? Well, it's this example. It's a bad example. I mean, this one about the BULLETHOLE, this one with the KISSES is different. I would need to say that..... The first X, the first PARAMETER..... Does something actively, to the second PARAMETER. But all I've got is BULLETHOLE.

E: How do you mean does something actively?

(4) Well, in the example it's got KISSES which is an active thing. HAS BULLETHOLE.... HAS BULLETHOLE.... Shall I try, HAS, no....

In the final segment, S5 is seen making a comparison of her second program with the INFECT program (1). It is somehow endearing of her to suspect that there is something wrong with INFECT-80, instead of there being a fault in her own program (2). In (3) S5 returns to her concerns voiced in the previous extract: It's this bullethole business, isn't it?" (= 'HAS-BULLETHOLE' hasn't convinced SOLO that it is a verb.) Finally, (4) the Subject spells out exactly what the misconception is, and the experimenter was able to understand S5's hour long insistence that there had to be a verb.

Since S5 had abstracted a faulty model of recursion from her original study of INFECT-80, it is perhaps not too surprising that all she discovered about recursion in subsequent study of the program was a reflection of her own faulty model. She ritualistically used this internalized mal-model to guide the design of one program after another, only to be frustrated time and again in the unpredictable nature of the output of these various solutions to the 'BULLETHOLE' problem.

8.5 A PRODUCTION SYSTEM MODEL OF AN IMITATION STRATEGY.

The PS presented in this section is an 'ideal' imitation strategy. The purpose of designing an ideal imitation strategy was so that its trace could be compared with the behaviour of individuals who use the strategy in devising their programs. Of the six Subjects studied in depth, five imitated programs from the Programming Manual at least part of the time they spent on the 'BULLETHOLE' problem, and four of the Subjects used the imitation strategy extensively. In the PS (ideal Imitation) only one extra rule has been added to model a single aspect of S5's behaviour: the rule which restricts the relation slot of the triple embodying the wild-card pattern match to be 'an active relation'. Although no further aspects of S5's behaviour has been modelled, the ideal imitation strategy provides a good starting point for analyzing in detail the operation of this particular strategy by examining variations from the ideal strategy. The reader may like, as a final exercise, to examine again the S5 protocol given above in comparison with the rules of the PS provided in the remainder of this Chapter.

The following production system uses three external memories:

(i) the problem-statement, labelled [problem-statement];

(ii) the vdu, labelled [vdu];

(iii) the course unit, labelled [course-unit].

The LHS functions are:

BADLINEP :- retrieves the first erroneous line in the program.

ACTIVEP :- returns 'true' if its argument is an active verb.

CORRESPONDINGLINEP :- is true for its first argument, if the first argument matches the second.

The RHS functions are:

SUBST :- this function deletes its third argument from working memory
and inserts its first argument.

REHEARSE :- this function merely reasserts an element in working memory (i.e. does a *subst* of an element for itself).

FIND-EXEMPLAR :- this function performs the magical feat of finding an appropriate program to imitate in the course unit (in the current implementation it simply returns the INFECT procedure in the format shown below).

```
((to infect x)
 (1 note x has flu)
 ((2 check x kisses ?)
  (2a IP: infect * : exit)
  (2b IA: exit)))
```

COURSE-UNIT :- just changes the [course-unit] into the argument of this function. For example, if the course unit was opened at page 80, it might look something like: '([course-unit] pg80 .....)'. Now, the following call to the function '*course-unit*' would replace this with '([course-unit] pg90 .....)':

```
(fool (([course-unit] pg80 .....)) ==> ((*course-unit* (pg90 .....))))
```

RUN :- runs the program, and adds SOLO's output to the display '[vdu]' in the appropriate format, i.e. in a form which can be used by rule IR4 for comparison with the expected output.

EDIT :- changes part of the display ([vdu]).

MISMATCH :- returns the difference between its two arguments, (e.g. (*mismatch* (CHECK /X/ SHOOTS ?) (CHECK /X/ KISSES ?)) gives SHOOTS).

At the start of execution of the SHOOTUP problem, working memory is assumed to be as follows:

```
((goal: achieve program)
 (expected-output: (sandwich plate newspaper book table floor
                   (has bullethole)))
 (chain-rel: (on (sandwich plate newspaper book table floor)))
 (procedure-name: shootup))
```

The external memories are as follows:

```
([course-unit])

([problem-statement] shoots bullethole on)

([vdu])
```

Rule IR9mc is tailored specifically for S5. The others represent idealised imitation rules:

```
(defv imitation-rules (
(IR1  ((program correct))
      ==>
      ((*halt*)))

(IR2  ((program achieved))
      ==>
      ((goal: test program)))

(IR3  ((goal: test program)
       (chain-rel: (on (=firstnode &)))
       ([vdu] &lines (TO =procname =var) &rest))
      ==>
      ((*run* =procname =firstnode)
       (goal: compare expected output with actual output)))

(IR4  ((goal: compare expected output with actual output)
       (expected-output: &output)
       ([vdu] & &output &))
      ==>
      ((program correct)))

(IR4b ((goal: compare expected output with actual output)
       (expected-output: &output)
       ([vdu] & (*not* &output) &))
      ==>
      ((program incorrect)))

;IF the goal is to imitate a program, and there is a program to imitate,
;THEN set up a pointer to the top line of the program, and set up a goal
;     to imitate the predicate.
(IR5  ((goal: imitate program)
       ([course-unit] exemplar: =topline &))
      ==>
      ((exemplar-pointer: =topline)
       (goal: imitate predicate)
       (goal: instantiate predicate)))

(IR6  ((goal: imitate predicate)
       (exemplar-pointer: =current-predicate)
       ([course-unit] exemplar:
                      & =current-predicate =next-predicate &rest))
      ==>
      ((*subst* (exemplar-pointer: =next-predicate)
                for
                (exemplar-pointer: =current-predicate))
       (*rehearse* (goal: instantiate predicate))))


(IR7  ((goal: imitate program)
       ([course-unit]))
      ==>
      ((*subst* ([course-unit] exemplar: (*find-exemplar*))
                for
                ([course-unit]))))
```

```
(IR8  ((goal: achieve program))
      ==>
      ((goal: imitate program)))

;IF the goal is to instantiate the predicate to be imitated,
;   and that predicate is of the form 'TO ...',
;THEN write that line, substituting the new procedure name for
;     that in the exemplar.
(IR9a ((goal: instantiate predicate)
       (exemplar-pointer: (to = =var))
       (procedure-name: =procname))
      ==>
      ((*edit* (to =procname =var)))

;IF the goal is to instantiate the predicate to be imitated,
;   and that predicate is of the form 'NOTE .....',
;THEN write that line in, substituting the new relation and second node
;     for that in the exemplar.
(IR9b ((goal: instantiate predicate)
       (exemplar-pointer: (=n note =var = =))
       (expected-output: (& (=rel =node2)))
      ==>
      ((*edit* (=n note =var =rel =node2)))

;IF the goal is to instantiate a predicate,
;   and the predicate is a 'CHECK',
;THEN copy the check-line, inserting the relation relevant to the
;     current program, and insert the new procedure name.
(IR9c ((goal: instantiate predicate)
       ([course-unit] exemplar: ((to =example-procname ) ))
                                                   ← ←
       (exemplar-pointer: ((=n check =var = ?)
                           (=a IP: =example-procname rest1)
                                   ←
                           (=b IA: rest2)))
                ←
       (chain-rel: (=rel ))
                    ←
       (procedure-name: =procname))
      ==>
      ((*scratchpad* ((=n check =var =rel ?)
                      (=a IP: =procname rest1)
                      ←
                      (=b IA: rest2))))
            ←

;The following 'mal' rule version of IR9c is tailored to S5:
;IF the goal is to instantiate a predicate,
;   and the predicate is a 'CHECK',
;THEN copy the check-line, inserting an active relation in the
;     CHECK part of the current program.
(IR9mc        ((goal: instantiate predicate)
       ([course-unit] exemplar: ((to =example-procname &) &))
       (exemplar-pointer: ((=n check =var = ?)
                           (=a IP: =example-procname &rest1)
                           (=b IA: &rest2)))
       ([problem-statement] & (*activep* =rel) &)
       (*not* (bad: =rel))
       (procedure-name: =procname))
      ==>
      ((*edit* ((=n check =var =rel ?)
                (=a IP: =procname &rest1)
                (=b IA: &rest2)))))

;IF the goal is to instantiate a predicate,
;   and that predicate is 'DONE',
;THEN write 'DONE', and note that the program has been achieved.
(IR10 ((goal: instantiate predicate)
       (exemplar-pointer: done))
      ==>
      ((*edit* (done))
       (program achieved)))))

(IR11 ((program incorrect)
       ([vdu] &w (*badlinep* =badline) &x)
       ([course-unit] &y (*correspondinglinep* =line =badline) &z))
      ==>
      ((*subst* (exemplar-pointer: =line)
                for
                (exemplar-pointer: &))
       (bad: (*mismatch* =badline =line))))


     ←
```

CHAPTER 9

EPILOGUE

9.1 ACHIEVEMENTS.

In the pages of this thesis research into the behaviour of novice programmers has been reported. The focus of attention has been on the knowledge novices possess, and the way they deploy the knowledge they have acquired in solving programming problems. This approach represents a major shift away from the current paradigm, which has been content merely to demonstrate that there are differences between novices and experts, and to characterize the differences mainly in terms of the knowledge the expert has. The experiments reported in Chapters 2, 4, and 5 demonstrate that interesting, relevant, and informative tasks can be designed for studying novice programming behaviour. These tasks help us specify in some detail what novices know, and provide us a basis for comparison with what the expert knows. There are any number of tasks which novices routinely perform which could and should take the place of the laboratory tasks which have been used in the past to study novice programmers.

An 'interactionist' theory of problem solving was presented. The theory posits a role for both domain related and domain independent Knowledge in problem solving. The Interpretation Theory, which embodies the theory of problem solving, has been useful in determining the overall structure of the behaviour of the Subjects who were studied. S8's protocol was not wildly at variance with the theory.

The major weakness of the protocol analyses reported in Chapters 7 and 8 is that there was no independent scoring of the protocols. In future, all protocols should be so scored, with a minimum 80% agreement on the identification of important categories of response (categories 1 - 4) before the protocols are used as support for the theory. Anything less than 80% agreement should be regarded as a disconfirmation of the theory.

Also, there is not a one to one mapping between the problem solving model incorporated in the Interpretation Theory and the statements in the think-aloud protocols, which indicates that the Interpretation Theory needs to be carefully re-examined.

The theory also states that there are no differences between novices and experts in their general approach to solving a given problem. This general approach (according to the theory) involves the selection, application, and evaluation of domain related and domain independent Knowledge structures, with respect to elements of the particular problem in hand.

Although there is some support for the theory in the protocols, rigorous tests need to be made in an effort to disconfirm the theory. It may be that the theory is viable only in circumstances where fairly integrated, domain related knowledge structures already exist in the minds of programmers.

Expertise in some domain is reflected in the 'early selection' in the problem solving process, of 'domain intelligent' schemas which, once activated, assume a predominating role in guiding subsequent processing. A domain intelligent schema is defined as one which encodes both declarative and process information, the latter of which alone permits application of domain relevant knowledge to the solution of problems. A novice is a problem solver who has acquired 'more or less' intelligent, domain related schemas. For example, some novices develop robust, but inaccurate, mental models of the behaviour of recursive procedures, and are able to use these models in the same way as experts use their knowledge of recursion in solving problems. The difference between such novices and experts is that there are problems which these novices will not be able to solve, since their knowledge is inaccurate. Other novices seem to have acquired little more than a 'label' for recursion. That is, their developing recursion schema contains little more than an index to the label and a pointer to the section of the Programming Manual which discusses recursion.

This is reflected in the overwhelming reliance on an imitation strategy by the majority of the novices who attempted the 'BULLETHOLE' problem. Five out of the six Subjects (S6 through S10) used the strategy. One of the goals for the immediate future is to implement an improved Imitation strategy, and to incorporate some learning processes in it. A couple of the protocols (which have not been presented in this thesis) show that some novices are capable of 'learning by doing': they try something out, carefully evaluate the effect of the trial, and store the result for later use. Others are good only at trying things out, but seem to be deficient in evaluation rules. Something either works or it doesn't work. This sort of approach was displayed repeatedly by S5, who would code one program after another, and then simply type the program in at the terminal and run it without any certain expectations about its outcome.

Also, there are some interesting indications that some novices are very good at performing a wide range of tasks and that some others are consistently not very good performers. The same seems to hold true for the experts we used as a standard of comparison against the novices, in whom we were mainly interested. For example, one expert looked quite like a novice on both the Questionnaire task, and on the Transcription task. One novice performed like an expert on the Transcription task, and was good at understanding a programming problem and in designing and evaluating a program. This is an interesting set of findings that should and will be followed up.

Finally, the model has to be extended. A wider range of concepts must be investigated and a wider range of programming languages. Both of these latter tasks are already in hand.

APPENDIX A

PROGRAMS AND PROTOCOL GIVEN BY S1 ON THE 'GUILTY BY ASSOCIATION' PROBLEM.

```
TO IMPLICATE /X/
1 CHECK /X/ IS A BURGLAR
1A IF PRESENT: NOTE "/X/ IS A CROOK"; CONTINUE
1B IF ABSENT: PRINT "/X/ IS IN THE CLEAR "; EXIT
2 CHECK /X/ WORKS FOR ?
2A: IF PRESENT IMPLICATE *
2B: IF ABSENT: EXIT
DONE


TO IMPLICATE /X/
1 NOTE /X/ IS A CROOK
2 CHECK /X/ WORKS FOR ?
2A IF PRESENT IMPLICATE *; CONTINUE
2B IF ABSENT: EXIT
3 CHECK * WORKS FOR ?
3A IF PRESENT IMPLICATE *
3B IF ABSENT: EXIT


TO IMPLICATE /X/
1 CHECK /X/ IS A BURGLAR
1A IF PRESENT NOTE /X/ IS GUILTY; CONTINUE
1B IF ABSENT: PRINT X IS NO CRIMINAL; EXIT

TO GUILTTEST /X/
1 CHECK /X/ WORKSFOR ?
2A IF PRESENT: NOTE "* IS A CROOK"; EXITr
2B IF ABSENT: PRINT "* IS IN THE CLEAR"; EXIT

2 CHECK /X/ WORKS FOR ?
2A IF PRESENT
```

1: ex: So we just have to do the assignment.

2: That's right.

3: Which I've just made an awful mess of apart from the fact that it's been caught in the rain.

4: ex: Well, that's quite all right.

5: I just couldn't get anywhere with this.

6: I mean I thought I quite confidently understood the pages in the book related to recursion but what I couldn't relate was how I could build...

7: I was....

8: I was assuming 'if someone is found to be guilty then whoever that person works for is also guilty.'

9: I was assuming that I couldn't start off by NOTEing that Liddy was guilty.

10: I was assuming that I had to make a CHECK to tell me that Liddy was guilty.

11: So I thought that was easy enough.

12: ex: Is that.... Did you try working it out a bit at a time as you've just described, or did you try to work it out all at one go? Getting a general impression of what you had to do or did you in fact as you just said start off with Liddy?

13: Well, I read all that through first and that all seemed quite straight- forward in terms of implicating somebody if somebody else works for them....

14: I mean that's a funny way round.

15: That part of it I understand, but what I couldn't work out was how I could start off the procedure to establish that liddy was in fact guilty, right?

16: I mean I started off: TO IMPLICATE: CHECK X ISA BURGLAR; IF PRESENT: NOTE X ISA CROOK; CONTINUE.

17: ex: Why 'X is a crook'?

18: Well, I just meant a change from saying 'GUILTY'.

19: ex: But it's equivalent?

20: That's just me trying to feel as if I had some input of my own into it, right?

21: IF ABSENT: PRINT X IS IN THE CLEAR, is not guilty.

22: X is....

23: That would be the end of the procedure.

24: Then on the next step I was trying to run a CHECK on who Liddy works for and then this is where my recursion procedure came unstuck because if I then put IF PRESENT: IMPLICATE wild-card Y it goes right back to Step 1 of the procedure and....

25: Nobody else is a burglar.

26: The reason they're guilty is because the person works for them.

27: So what I couldn't work out was a way of just not going back to the beginning of the procedure but just to take a part of it.

28: So then I thought 'Aha! What I need is a subprocedure, you see, and this is where I really got messed up.

29: In fact I mean it just went to pieces.

30: For one moment I thought I saw the light and I was going to have a subprocedure called TO GUILTTEST I called it because it couldn't be TO IMPLICATE.

31: So what did I think I was doing there?

32: Well, CHECK X ISA BURGLAR, that was the same....

33: And then I went back to using 'guilty', I was (laughter and something unintelligible).

34: All these smart definitions and I couldn't even work it out.

35: I thought I'll just stick to the basics.

36: (mumbling)

37: And then what I think I was going to do was say CHECK X WORKS FOR so-and-so, IF PRESENT: GUILTTEST asterisk.

38: But then I found I was just duplicating the same part without actually getting to the different definition that I wanted.

39: I thought I tried something else.

40: At that point I just....

41: I just couldn't see what I was meant to be doing any more.

42: And then I thought 'Ah! Perhaps I misunderstood. Perhaps I made it more complicated than I need to. Perhaps you don't need to establish Liddy's guilt. Perhaps you can just assume it.'

43: But then I read on....

44: Well, I'd already read through this before, sorry....

45: To me, this bit here about 'you may want to do something more elaborate, for instance you may want to include extra CHECKs to see if other con- ditions are met before someone is implicated, e.g. is that person a known criminal, etc.'

46: I didn't see the need to do that when the definition of the procedure
     only makes the following inference: if someone is found to be guilty
     then whoever that person works for is also guilty.

47: So I didn't see the point of running....

48: Whether or not they're a criminal I thought was irrelevant to the procedure of IMPLICATE.

49: They're implicated by dint of who they work for.

50: And I wondered whether I was meant to go through in each stage putting CHECK X ISA BURGLAR, CHECK X ISA BIGLAWYER, CHECK X ISA PRESIDENT.

51: ex: you thought of doing that?

52: Yes.

53: ex: And did you actually?

54: No, I didn't because I thought, well, a burglar is a criminal, but a biglawyer isn't necessarily a criminal.

55: Nor is a president.

56: ex: So what situation are we in at the moment?

57: Well, I'm completely all at sea now.

58: ex: It is not permissible to be all at sea.

59: Oh....

60: I have to do something?

61: ex: When you came here did you think you were defeated on this?

62: Yes.

63: ex: Well, which way can you go now?

64: The only thing I feel I can possibly do, which I don't think is being asked of me is to....

65: By NOTEing that Liddy is guilty, I then think I can put in the rest of the procedure which will work through the rest of these other characters okay, but I....

66: That's cheat.....

67: That's cheating, well, that's cheating.

68: ex: Why is that cheating?

69: Because I haven't CHECKed whether Liddy in fact is guilty.

70: I've just NOTEd the fact he's guilty and from thence implicated.

71: ex: And you're definitely required to check that he's a burglar or something first?

72: Yes.

73: So your problem is, having done that, noting he's guilty, getting the others?

74: Well, I think that is quite straightforward.

75: ex: What? Getting uhh....?

76: Getting the others once I've got Liddy.

77: ex: Ah. I see. How are you;going to get the others once you've got Liddy?

78: I'm going to CHECK who works for who.

79: ex: And do what?

80: And IMPLICATE them.

81: ex: And Implicate does what?

82: Well, IMPLICATE infers that someone is guilty.

83: ex: If they're a burglar. I mean you first....

84: Oh, on my....

85: ex: You came back. Well, in fact you came back to that and said.... This is your second guess, right? But originally the procedure you've just come back to you said that Implicate involves checking first of all to see if Liddy's a burglar and then the rest of it's very simple and it's simple because you just see if they work for someone and come back to Implicate and of course when you come back to Implicate what's the first thing you have to do?

86: ex: To CHECK whether someone's a burglar.

87: ex: So it's not as simple as you thought?

88: No, sorry, I wasn't actually talking about that one.

89: I was talking about the one where I assumed from the start that Liddy...

90: That Liddy was a burglar.

91: I see. I see.

92: And then I could....

93: I was confident that I could CHECK the working relationships between the others from that....

94: No, I can't see any way around that, I'm afraid.

95: Can you give me a clue?

96: ex: It's not in the nature of the game actually.

97: All right, well, I mean....

98: ex: I mean I'd like to see you solve it. I'd like to see.... I mean it's interesting so far, the types of solution you've tried.

99: But I mean I have thought about it such a lot and I've just got myself into such a muddle now that I know the answer's....

100: I'm sure the answer's quite straightforward because everything so far has been quite straightforward and it's more or less directly related to the part of the course it says it has.

101: I think here it's asking me to make some kind of jump that I haven't been shown how to do before and I can't seem to grasp hold of it.

102: ex: Well, what are your options? Which way would you like to try to do it? I mean if you had your way what would you like your procedure to do?

103: Well, it would have to assume, I mean, it would have to NOTE that Liddy was guilty to start, 'cause that's the only procedure I've worked out that will....

104: That will give the required answers.

105: ex: Well, could you try writing a procedure? Actually, I'd like to have a procedure before this is over. I mean if you were doing the course you'd eventually have to come down on some side or other and do something.

106: Well then, all I'd be able to do is this....

107: This simple procedure here.

108: ex: Well do you want to try it and let's see what happens?

APPENDIX B

THE FULL TEXT OF THE INSTRUCTIONS GIVEN TO SUBJECTS ON THE TRANSCRIPTION TASK, CHAPTER 5.

On the table in front of you there is a booklet and a set of coloured pens. The booklet contains a number of passages of text. All I want you to do is to copy all the passages, word for word, after the procedure has been explained to you. First of all, the booklet contains all the passages. Each passage has a blank sheet of paper in front of it and two blank sheets of paper behind it. The sheet of paper in front of each passage is there to prevent anyone from seeing the passage before they've received a signal to look at it. The first sheet of paper behind each passage is a dark colour, and is there to prevent anyone seeing the passage through the back of the page it's written on when the page is turned. This is not to suggest that anyone would think of cheating. It's just that when I made up the first booklets, I discovered that the text could be seen through the back of the paper, so I had to make up another set of booklets with the extra sheet between the page with the text on it, and a page for Subjects to write on. The white page behind the coloured page is what I want you to write on. Let me show you what I mean. [At this point, the experimenter showed each Subject that by lifting the first page of the booklet, a passage of text would be revealed, and that on the second page following the text there was a white sheet of paper for writing on.] You also have four different coloured pens and a pencil, laid out in the order: red, black, blue, and green pens, and then the pencil. What I want you to do is this: when I give you a signal, turn over the first page and look immediately at the passage of text written on the page in front of you. Have the red pen in your hand, ready, so that when you start writing you won't have to search around for the pen. When you turn the page to look at the text, I'll start up this stopwatch and time you for ten seconds. When the ten seconds are up, I'll say 'Stop'. When I say 'Stop', I want you to turn over to the second page behind the passage you've been reading, and write down everything you can recall of what you've just read. You can have as long as you like to write down your recall. When you've recalled everything you can, I want you to turn back to the text, and have a second go. You can turn back to the text whenever you are ready. I'll be watching you, and as soon as you turn back, I'll restart the timer, and give you another ten seconds to read before saying 'Stop'. When I say 'Stop', you write what you can recall the second time. Altogether, you'll get five opportunities to read each passage, with ten seconds allowed each time. Are you right handed? [All Subjects responded 'Yes'.] Okay. You can make it easy for yourself by finding the page you have to write on before looking at the passage you have to read, and keep the intervening page and the page with the text on it together in your left hand, like this. [Experimenter showed each Subject how to make himself ready to turn to the writing page with the minimum of fuss.] That way, you can keep the pages together in your left hand while both reading and writing and not be distracted looking for the correct page in-between times. Also, after writing down all you can recall, put the pen down at the far right of the line of pens, and take the next one before you go back to read a passage again. That way you won't forget what you've just read while you're looking for the correct pen to write with. If I see you start to read or re-read without a pen in hand, or with the wrong pen, I'll say so. Okay. Now, the idea of this task is to extract as much information as possible from the passages you are going to read, and to do it as quickly as possible, but you also have to be very accurate. If, for example, the first line of text you happen to read has six words on it, then I want that fact reflected in what you write down. That is, the first line you write should have exactly six words. If the first and third and fifth words of that line happened to be capitalized, then you should capitalize these words when you write down what you are able to recall. In other words, I'd like to be able to say, at the end of the experiment, that what you've written perfectly reflects everything in all the passages you were given to read. Now, although you have five 'goes' at each of these passages, it would be good if you could copy each passage in fewer goes. If you could copy the whole text after the first view, that would be excellent. What I want to know is how much information can be extracted from a passage of text such as those you're going to see. What I want you to do is to put down everything you can recall. If you think you can recall something, but you're not sure, put it down anyway. If you're wrong, you can always cross a line through errors on subsequent goes. If you think you can guess about the contents of what you've read but can't remember, do that. If you're wrong, you can always cross it out when and if you discover an error. Don't be afraid of making errors. What matters is that it's all correct in the end. When you write down your recall, it's a good idea to leave a blank line between each line you write so that you will have room to correct any errors you might discover on subsequent readings and recalls. If you do discover an error, just put a line through the word or group of words that are wrong, and write in the correct word or words. Don't scribble over an error, as I'd like to be able to look at them after the experiment is over. One final thing. Although I would like you to copy out each passage in the fewest possible number of goes at it, you must read each passage five times. It won't make any difference to your final score, because if you've managed to copy the whole thing after three goes, I'll be able to see that because the whole passage will be copied in red, black, and blue. The other two goes would give you an opportunity to check and recheck what you've written, that's all. And you must also take the full ten seconds for reading every time you return to a passage. In other words, once you've started reading you can't stop and start writing until I say 'Stop'. Okay, that's it. Do you have any questions about what I want you to do? [Any queries are dealt with at this point. During the first trial all Subjects inevitably begin by forgetting to start a reading phase with the correct pen in hand, but by the end of the first trial they become quite adept at following the procedure.]

REFERENCES

Adelson, B. Problem solving and the development of abstract categories in programming languages. Memory and Cognition, 1981, 9, 422-433.

Anderson, J.R. Language, Memory and Thought, Lawrence Erlbaum Associates: New Jersey , 1976.

Anderson, J.R. Cognitive psychology: its implications. New York: Academic Press, 1980.

Atwood, M.E., & Ramsey, H.R. Cognitive structures in the comprehension and memory of computer programs: an investigation of computer debugging. Technical Report TR-78-A21, U.S. Army Research Institute for the Behavioral and Social Sciences, Alexandria, VA, 1978.

Barr, A., Beard, M., & Atkinson, R.C. The computer as a tutorial laboratory: The Stanford BIP Project. International Journal of Man-Machine Studies, 1976, 8, 567-596.

Bhaskar, R.& Simon, H.A. Problem solving in semantically rich domains: an example from engineering thermodynamics. Cognitive science, 1, 193-215, 1977.

Bobrow,D.G. & Norman, D.A. Some principles of memory schemata. In D.G. Bobrow & A.M. Collins (Eds.), Representation and understanding: studies in cognitive science. New York: Academic press 1975.

Bransford, J.D., - Johnson, M.K. Contextual prerequisites for understanding some investigations of comprehension and recall. Journal of verbal learning and verbal behaviour, 11, 717-26. 1972.

Bransford, J.D., & Johnson, M.K. Considerations of some problems of comprehension. In W.G. Chase (ed.) Visual Information Processing. Academic Press, New York, 1973.

Breuker, J. Availability of Knowledge. COWO - publicatie 81-JB. 1981.

Brooks, R. Towards a theory of the cognitive processes in computer programming International Journal of Man-Machine Studies, 1977, 9.

Byrne, R. Planning meals: problem solving on a real data base. Cognition 5, 287-332. 1977.

Card, S.K., Moran, T.P., & Newell, A. Computer text editing: an information processing analysis of a routine cognitive skill. Cognitive Psychology, 1980, 12, 32-74.

Chase, W.G. & Simon, H.A. Perception in chess. Cognitive Psychology, 1973, 4, 55-81.

Chi, M.T.H., Feltovich, P.J. & Glaser, R., Categorisation & Representation of Physics Problems by Experts & Novices. Cognitive Science, 1979, 5, 121-152.

Collins, A., Brown, J.S. & Larkin, K.M. Inference in text understanding. In Spiro, R.J., Bruce, B.C., & Brewer, W.F. (Eds.) Theoretical Issues In Reading Comprehension, Lawrence Erlbaum Associates, New Jersey, 1980, 385-407.

Collins,A. & Gentner, D. Constructing runnable mental models. Proceedings of the Fourth Annual Conference of the Cognitive Science Society, Michigan, 1982.

Collins, A.M. & Loftus, E.F. A spreading-activation theory of semantic processing.Psychological Review. 1975, 82, 407-428.

Davis, R. & King, J.An overview of production systems. Stanford University, 1975.

Ehrlich, K. & Soloway, E.An emperical investigation of the tactic plan knowledge in Aprograming. Yale University, 1982.

Eisenstadt, M. Artificial intelligence project. Units 3/4 of Cognitive psychology: a third level course. Milton Keynes: Open University Press, 1978.

Eisenstadt, M. Design features of a friendly software environment for novice programmers. Technical Report No. 3, Human Cognition Research Laboratory, 1982.

Ericsson, K.A. & Simon, H.A. Verbal reports as data. Psychological Review. Vol. 87, No. 3, 1980.

Friedman, A. Framing pictures: the role of knowledge in automatized encoding and memory for gist. Journal of Experimental Psychology: General. Vol. 108, No. 3, 316-355. 1979.

Goldstein, I.P. Summary of MYCROFT: a system for understanding simple picture programs. Artificial Intelligence, 1975, 6.

Hayes, J.R. & Simon, H.A. Understanding written problem instructions. In Gregg, L.W. (Ed.), Knowledge and Cognition. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1974.

Hazemere, T. AURAC: a debugging aid for novice programming in SOLO. In O'Shea, T. (Ed.) New Technologies in Education. (in press).

Heller, J.I. & Reif, F. Analogical reasoning patterns in expert problem solving. Proceedings of the Fourth Annual Conference of the Cognitive Science Society./ Ann Arbor, Michigan, 1982.

Jeffries, R. A comparison of the debugging behaviour of expert and novice programmers. Paper presented at the AERA annual meeting, March, 1982.

Just, M.A. & Carpenter, P.A. A theory of reading: from eye fixations to comprehension. Psychological review, 87, 329-354, 1980.

Kahneman, D. Attention and Effort. Englewood Cliffs, New Jersey: Prentice-Hall 1973.

Kintsch, W. The representation of meaning in memory. Hillsdale, N.J: Lawrence Erlbaum Associates, 1974.

Larkin, J., McDermott, J., Simon, D., & Simon, H. Expert and novice performance in solving physics problems. Science, 208. 1980.

Lewis, M. Improving SOLO's user-interface: an empirical study of user behaviour and proposals for cost-effective enhancements to SOLO. Technical report no. 7, Computer Assisted Learning Research Group, The Open University, 1980.

Mayer, R.E., The psychology of how novices learn computer programming, Computing Surveys, 13, 1, March 1981.

McDermott, J. & Forgy, C. Production systems conflict resolution strategies, Acedemic Press, 1978.

McDermott, J. Larkin, J.H. Re-representing textbook physics problems. Proceedings of the 2nd National Conference of the Canadian Society for Computational Studies of Intelligence. Toronto: University of Toronto Press, 1978

McKeithen, K.B., Reitman, J.S., Rueter, H.H., & Hirtle, S.C. Knowledge organization and skill differences in computer programmers. Cognitive Psychology, 1981, 13.

Nelson, K. Rescoria, L. &Gruenoes, J. Early lexicons: what do they mean? Child Development 49, 960-68. 1978.

Newell, A., & Simon, H.A. Human problem solving. Englewood Cliffs, N.J.: Prentice-Hall, 1972.

Norman, D.A. Steps toward a cognitive engineering: design rules based on analyses of human error. Proceedings of the Conference on Human Factors in Computing, Washington, D.C.: ACM Washington Chapter, 1982.

Norman, D.A. Some observations on mental models. In D. Gentner and A. Stevens (Eds.), Mental models. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1982.

Norman, D.A. & Bobrow, D.G. Descriptions: A basis for memory acquisition and retrieval. Technical Report 74, Center for Human Informtion Processing, 1977.

Novak, Jnr., G.S. Representations of knowledge in a program for solving physics problems. Proceedings of the 5th international joint conference on artificial intelligence. Massachusetts, 1977.

Palmer, S.E. Visual perception and world knowledge: notes on a model of sensory-cognitive interaction. In D.A. Normen, D.E. Rumelhart

& the LNR research group. Explorations in cognition. San Francisco: Freeman, 1975.

Reitman, J.S, & Rueter, H.H. Organization revealed by recall orders and confirmed by pauses. Cognitive Psychology, 12, 4, 554-581, 1980.

Rich, C. & Shrobe, H. Initial report on a Lisp programmer's apprentice. IEEE Trans. on Software Eng. Vol.4, No. 5 1978.

Rumelhart, D.E. Notes on a schema for stories. In D.G.Bobrow & A. Collins (Eds). Representation and understanding: studies in cognitive science New York: Academic press. 1975.

Rumelhart, D.E. & Ortony, A. The representation of knowledge in memory. Technical report 55, Center for Human Information Processing, 1976

Ruth, G.R. Intelligent program analysis. Artificial intelligence, 7, 1976.

Schank, R.C. Identification of conceptualizations underlying natural language. In R.C. Schank & K.M. Colby (Eds.), Computer models of thought and language. San Francisco: Freeman, 1973.

Schank, R.C. The structure of episodes in memory. In D.G.Bobrow & A.M. Collins (Eds.), Representation and understanding: studies in cognitive science. New York : Academic press, 1975.

Schank, R.C. How much intelligence is there in artificial intelligence? Intelligence 1980, vol.4, No. 1, 1-14.

Schank, R.C. & Abelson, R. Scripts, plans, goals and understanding. Lawrence Erlbaum Associates, Hillsdale New Jersey. 1977.

Schank, R.C.- Riesbeck, C.K. Inside computer understanding: five programs plus minatures. Lawrence Erlbaum Associates, Hillsdale New Jersey. 1981.

Schoenfeld, A.H. Teaching problem-solving skills. American Mathematical Monthly, 1980, 87, 794-805.

Sheil, B. The psychological study of programming. Computing surveys vol. 13 No. 1, 1981.

Shneiderman, B. Software psychology: human factors in computer and information systems. Cambridge, MA: Winthrop, 1980. B

Shrobe, H., Waters, R., & Sussman, G. A hypothetical monologue illustrating the knowledge underlying program analysis. MIT Artificial Intelligence Laboratory Memo 507, 1979.

Sime, M.E., Green, T.R.G., &Guest, D.J. Psychological evaluation of two conditional constructions used in computer languages. International Journal of Man-Machine Studies, 5, 1, 123-143. 1973.

Simon, H.A., Models of Thought, Yale University Press, 1979. Simon, H.A. & Gilmartin, K.J. A simulation of memory for chess positions. Cognitive.Psychology 5, 29-46. 1973.

Soloway, E., Ehrlich, K., & Bonar, J. Tapping into tacit programming knowledge. Proceedings of the Conference on Human Factors in Computing, Washington, D.C.: ACM Washington Chapter, 1982(a).

do novices know about programming? In B. Shneiderman & A. Badre (Eds.) Directions in human-computer interactions. Norwood, N.J.: Ablex, 1982.

Sussman, G.J. A computer model of skill acquisition. New York: American Elsevier, 1975. Thibadeau, R., Just, M.A. & Carpenter, P.A, A model of the time course and content of reading, Cognitive Science, 6, 157-203, 1982.

Wason, P.C. Hypothesis testing and reasoning. Cognitive psychology, D303, block 4, unit 25. Open University Press: Milton Keynes. 1978.

Wason,P.C. & Evans, J. St B.T. Dual processes in reasoning? Cognition 3, 141-54. 1975.

Waters, R.C. A method for analysing loop programs. IEEE Trans. on Software Eng. vol. SE-5, No. 3, 237-247, 1979