Knowledge restructing and the development of expertise in computer programming

Davies, Simon P.

KNOWLEDGE RESTRUCTURING AND THE DEVELOPMENT OF EXPERTISE IN COMPUTER PROGRAMMING

SIMON P. DAVIES

Department of Psychology

University of Durham

Thesis submitted in partial fulfilment of requirements for PhD in Psychology,
July, 1992.

Preface

The work reported in this thesis owes much to the support and understanding of my colleagues both past and present. A great deal of the groundwork for the thesis was established while I was employed by the University of Durham on a research project funded by IBM UK. A lot of the experimental work and much of the analysis was conducted at Huddersfield Polytechnic which provided an ideal environment for data collection. Although many of the experiments had already been written-up in one form or another, bringing out the links and connections between the individual experiments proved to be a difficult task.This was only really made possible at the expense of other things that I should have been doing at that time.

Foremost, I would like to thank my supervisor, John Findlay, for all his help and encouragement over the years. I would also like to thank Adrian Castell for the discussions we have had on cognate subjects. The final shape of this thesis, and many of the ideas expressed in it owe much to these discussions. Other people have also been instrumental though they may not realise it. Thomas Green's work originally inspired me to tackle programming and his encouragement over the past few years has sustained my interest. I would also like to thank David Gilmore for his patience over the last several months. Other people deserve thanks for other things. My parents for putting up with me, without putting me down. My friends for living with my sometimes strange (and usually bad) habits. My Grandfather, Cliff, who was a block off the old chip. Abbey Archer who has reduced my cynicism to at least a tolerable level, and to whom this work is dedicated. And, last but not least, the kids in my life - Boo, Kelly, Laurie and Thomas who have been inspirational in lots of ways.

Abstract

This thesis reports a number of empirical studies exploring the development of expertise in computer programming. Experiments 1 and 2 are concerned with the way in which the possession of design experience can influence the perception and use of cues to various program structures. Experiment 3 examines how violations to standard conventions for constructing programs can affect the comprehension of expert, intermediate and novice subjects. Experiment 4 looks at the differences in strategy that are exhibited by subjects of varying skill level when constructing programs in different languages. Experiment 5 takes these ideas further to examine the temporal distribution of different forms of strategy during a program generation task. Experiment 6 provides evidence for salient cognitive structures derived from reaction time and error data in the context of a recognition task. Experiments 7 and 8 are concerned with the role of working memory in program generation and suggest that one aspect of expertise in the programming domain involves the acquisition of strategies for utilising display-based information. The final chapter attempts to bring these experimental findings together in terms of a model of knowledge organisation that stresses the importance of knowledge restructuring processes in the development of expertise. This is contrasted with existing models which have tended to place emphasis upon schemata acquisition and generalisation as the fundamental modes of learning associated with skill development. The work reported here suggests that a fine-grained restructuring of individual schemata takes places during the later stages of skill development. It is argued that those mechanisms currently thought to be associated with the development of expertise may not fully account for the strategic changes and the types of error typically found in the transition between novice, intermediate and expert problem solvers. This work has a number of implications for existing theories of skill acquisition. In particular, it questions the ability of such theories to account for subtle changes in the various manifestations of skilled performance that are associated with increasing expertise. Secondly, the work reported in this thesis attempts to show how specific forms of training might give rise to the knowledge restructuring process that is proposed. Finally, the thesis stresses the important role of display-based problem solving in complex tasks such as programming and highlights the role of programming language notation as a mediating factor in the development and acquisition of problem solving strategies.

Contents List

FIGURES

Chapter 2

Chapter 3

Chapter 4

Chapter 7

Chapter 8

Chapter 1. Introduction and overview

The work reported in this thesis represents an attempt to understand the problem-solving processes involved in programming. My original motivation for studying programming arose from rather practical considerations i.e., how we might best design tools to support the problem-solving activities that are typically involved in programming tasks. However, in order to do this one needs to understand the nature of these problem-solving activities. Moreover, it rapidly became clear that existing theories of problem-solving do not provide an adequate account of the kinds of problem-solving behaviour that have been observed in programming. Hence, the original emphasis of this research moved away from the design of tools towards a more empirically motivated mode of research from which I attempted to gain some theoretical insight into the complexities of problem-solving in this domain.

Alan Newell (1973) once claimed that cognitive psychology tends to be phenomenon-driven, in that the discovery of a new phenomenon (for instance, the visual icon) leads to an exhaustive exploration of all its possible ramifications by cognitive psychologists. To some extent we could also claim that cognitive psychology and cognitive science are not only phenomena driven but also artifact driven. The widespread and continuing use of high-level programming languages has spawned a significant amount of research into the problem-solving processes involved in the creation and the comprehension of complex software artifacts. Much of the early research into the cognitive aspects of programming served simply to suggest that established theories developed in other problem-solving domains could be usefully applied to understand programming behaviour. However, more recently, research into programming behaviour has begun to contribute more explicitly to the general body of problem-solving theory, and has suggested modifications and important extensions to paradigmatic problem-solving analyses.

In trying to understand programming tasks it should be possible to frame an analysis in terms of existing generic models of problem-solving such as those proposed by Newell and Simon or by Anderson (described in the next chapter). These models are, after all, intended to be generic models of problem-solving, and not just models of problem-solving in specific domains. The work reported in this thesis adopts the general problem-solving paradigm proposed by these models. However, it is suggested that programming tasks display some characteristics that are not well catered for by such models. In the next chapter I will first present a brief overview of these generic problem-solving frameworks and their application, and then attempt to derive some common characteristics of these frameworks which will provide a starting point for assessing their validity as models which might be used to account for problem-solving behaviour in programming.

Preempting this discussion somewhat, I will suggest three major features of programming tasks that are not well represented by generic models of problem-solving. Firstly, programming places significant demands upon cognitive resources and programmers appear to be unable to elaborate an entire sequence of problem-solving steps without engaging in a closely linked series of planning and execution cycles. It will be suggested that existing models of problem-solving and planning do not emphasise sufficiently the close link between planning/operator selection and execution.

Secondly, existing models of human problem-solving typically describe situations in which problem-solving operators are applied directly to objects in the world (albeit, in many cases, an artificial symbolic world) - for instance, in solving puzzles or playing games. However, programming is rather different in that a programmer may use any one of a range of programming languages to implement a solution to a problem. Moreover, these languages display different kinds of characteristics which may or may not support a particular problem-solving approach.

Finally, in considering problem-solving in programming one needs to take account of the environment in which programs are typically constructed. For instance, compare writing a program using pen and paper with, say, dictating a

program (uncommon, but possible) or using a line-editor. Each of these environments has different characteristics, and these characteristics may, to a greater or lesser extent, support preferred problem-solving strategies.

A central issue raised by this thesis is that these features of programming tasks are not adequately represented by generic problem solving-models. In addition, it seems that these models or frameworks generally tend to emphasise aspects of a problem-solvers' knowledge representation and the way in which this determines strategy and have neglected to consider characteristics of the task or of the environment in which the task is undertaken. Studies of problem-solving in programming are now beginning to address these issues and have considered in some detail the way in which task characteristics, and in particular the notational features of languages, can affect problem-solving behaviour.

The work reported in this thesis attempts to link together findings that have emerged from studies of knowledge representation in programming and issues stemming from a consideration of the notational properties of programming languages. Until now these issues have been addressed in isolation. The central concern of this thesis is to provide a broader view and more detailed understanding of the problem-solving processes involved in programming. In particular, interest is directed towards understanding the way in which knowledge representation in programming develops, its relationship with different forms of programming strategy and the extent to which these are supported, or otherwise, by programming language and task characteristics.

The following chapter begins by outlining a number of generic theories or frameworks which have been advanced as models of human problem-solving. In fact, the main emphasis of this section is to provide a broad overview of human problem-solving from the perspective of two major theories or frameworks - namely, Newell and Simon's General Problem Solver (GPS) and Anderson's Adaptive Control of Thought (ACT *). There are clearly other frameworks and models which might provide an equally cogent account of human problem-solving. However, the application of GPS and ACT* has been extensive, and these frameworks have proven to be of great predictive value. Moreover,

while both frameworks provide a general description of the architecture of cognition, they differ substantiality in emphasis, and these differences provide important foci for a comparison of these theories.

The second chapter then moves on to consider more specific issues relating to problem-solving in programming. The intention of this section is to consider the common characteristics of problem-solving tasks and to illustrate why programming should be considered as a problem-solving activity. Subsequently, the application of generic problem-solving frameworks is considered in the context of programming. An attempt is then made to suggest that there are some important characteristics of typical programming tasks that are not well catered for by existing problem-solving models. While this chapter is broadly concerned with domain independent problem-solving frameworks, subsequent chapters provide a review of some of the more important empirical and theoretical work which has addressed more directly the problem-solving processes involved in programming.

Subsequent chapters are organised into three distinct themes - knowledge representation, strategy and notational features. The ease with which this emerged appears to reflect or belie the narrowness of traditional concerns in the psychology of programming. That is to say, very few studies actually attempt to suggest links between these different themes. This thesis represents a step in the other direction, and will hopefully provide a bridge between these various lacunae.

The third chapter considers expert-novice differences and knowledge representation, and reviews a number of studies originating in the programming domain. In the fourth chapter interest is directed toward understanding the nature of problem-solving strategies in programming, while chapter five considers the way in which programming language features (specifically notational factors) have been shown to affect problem-solving strategy. Each of the next six chapters reviews a specific experiment carried out to examine a particular hypothesis or a small collection of related hypotheses.

These empirical studies are reported in the order in which they were carried out, since initially no overarching model or framework existed which could be used to

neatly construe their theoretical development. However, as early studies in this series of experiments started to lead to the development of a model of problem-solving in programming, the theoretical focus of the work became clearer. It seems more appropriate to report the studies in this way rather than attempt to reconstruct the early experiments such that they fit with the model. While the early experiments in this series were generally exploratory, later studies make specific predictions stemming from the model proposed in this thesis. While allusion to the model is made in the context of reporting these experiments, the model itself is presented in the last chapter which attempts to draw together the results of the individual studies to suggest how features of a programmers knowledge representation develop and appear to interact with certain salient language features to determine particular forms of problem-solving strategy.

This model suggests a number of significant extensions to existing theories of problem-solving behaviour in programming, and in particular it addresses issues relating to the development of programming knowledge. The empirical work reported in this thesis illustrates how knowledge representation develops and how it changes with increasing programming expertise. Concern is directed primarily to exploring the adequacy of schema or plan-based theories of programming knowledge. The simple model of schema acquisition that is proposed by these theories to explain the development of expertise is challenged and a more complex model which emphasises knowledge restructuring processes is proposed.

This model suggests that knowledge representation is not uniform, but that as expertise develops certain features of a programmer's knowledge representation achieve prominence. For example, it is suggested that those aspects of a particular schema that directly encode the role of that schema will be more easily accessible than its other components. On this view, programming expertise is not seen to develop simply via the acquisition of a greater number and range of schemata, although this clearly is an important facet in the development of expertise. Rather, the emphasis of the model suggests the importance of knowledge restructuring, and empirical evidence for this phenomenon is cited.

For instance, in experiment five (reported in chapter 10) involving a program recognition task, it is shown that expert programmers respond to the presence of focal lines (i.e., those lines that directly encode the role of a particular plan or schemata) more quickly than they respond to non-focal lines. In the case of intermediates and novices no difference in response times to focal or non-focal lines is apparent. In addition, while intermediates and experts can detect plan violations with about the same frequency, experts detect such violations significantly faster than intermediates. These findings are taken as evidence for the idea that while both intermediates and experts are able to access the same range of plan structures, experts seem to be able to access the salient parts of these plan structures with great ease, perhaps suggesting that more experienced programmers structure their knowledge rather differently.

The mechanisms that give rise to this restructuring processes appear to be associated with a programmers design experience. The first experiment reported in this thesis suggests that design training may encourage programmers to think about problems in a more structured fashion, and may facilitate the construction of a mapping between the problem domain and structures in the language domain. It remains unclear whether design training and design methodologies simply reflect expert cognitive structure, and thereby describe or in some way facilitate naturally occurring problem-solving strategies or whether design training actually determines or predisposes a particular cognitive structure and set of related strategies.

In addition, the restructuring processes described by this model may suggest reasons for the adoption of particular forms of problem-solving strategy. In particular it may account for the transition from depth-first to breadth-first strategies that have been commonly observed in studies of the development of problem-solving expertise. Hence, expert programmers may develop a solution in a hierarchical fashion, starting from the implementation of lines of code that represent focal plan elements before expanding these to include subsidiary plan components. In the case of novices, and possibly intermediates, it is suggested that the uniform nature of their representation of plan knowledge does not allow for the differentiation of focal and non-focal plan elements and it is shown how

this might lead to the adoption of strategies which display depth-first characteristics.

Moreover, it is argued that features of certain programming language notations may provide support for the implementation or the comprehension of focal plan elements. For instance, languages such as Pascal have been described as "role-expressive" in that their rich lexical structure appears to facilitate the discriminability of the individual program components that might be expressed in that language. This has important implications for the model proposed here since we might predict that there will be an interaction between language structure (i.e., the extent to which a particular language might be described as "role-expressive") and expertise. Hence, we might expect experts to perform better using "role-expressive" languages such as Pascal where focal structures will be more readily perceived, while the performance of experts and non-experts in plan related tasks may not differ significantly in less "role-expressive' languages such as BASIC. Indeed, one of the experiments reported later in this thesis supports this prediction, and suggests that there is a strong interaction between language structure and expertise.

It will be suggested that the model of knowledge restructuring proposed here provides a parsimonious interpretation of a large body of existing empirical studies which have addressed problem-solving issues in programming. The model suggests ways of integrating previously distinct areas of research and indicates how aspects of a programmer's knowledge representation may interact with language and task features to determine programming strategy. In addition, the model has a number of distinct parallels with work in other domains, and this suggests that the model may be generalisable to other knowledge-intensive problem-solving tasks.

In summary, there are a number of identifiable aims relating to the work reported in this thesis. Firstly, an attempt is made to explore the cogency of extant problem-solving models or frameworks in terms of their application to programming. Secondly, the existing psychological work on problem-solving in programming is reviewed and evaluated. This review is grouped into three

sections - knowledge representation, strategy and language features - in order to reflect the thematic concern and historical development of this work. A number of experimental studies are then reported which address some of the shortcomings of existing work and provide the basis for a model of programming behaviour which attempts to bridge previously isolated areas of research, thus leading to a more unified framework for understanding programming behaviour. Finally, an attempt is made to relate this model to work in other problem solving domains and to illustrate briefly the cognitive mechanisms that might underpin the knowledge restructuring processes that are central to the model.

Chapter 2. Models of problem solving behaviour and their application to programming tasks

2.1 Introduction

This chapter provides a broad, yet necessarily brief, review of two important problem-solving frameworks, Newell and Simon's General Problem Solver (GPS) and Anderson's ACT*. The primary intention of this chapter is to examine the general applicability of these frameworks in terms of their ability to account for the complex problem-solving processes that underpin programming behaviour. It will be suggested that there are certain salient characteristics of programming tasks that are not well catered for by existing models of human problem-solving. This discussion provides both a basis for our characterisation of programming as a problem-solving activity and a foundation for the work reported later in this thesis.

Newell and Simon's problem and state space analysis of human problem solving is first introduced to provide a general background to the discussion presented in this chapter. This analysis is of some historical interest, but it appears to be of little significance in terms of its contribution to our understanding problem solving in the programming domain. This is because the focus of this work is upon relatively well-defined domains which involve little domain specific knowledge. This can be contrasted with the programming domain, where problems are generally ill-defined and where problem-solving activities are highly knowledge intensive. Despite the fact that the Newell and Simon model does not appear to be strictly relevant to our understanding of programming, it does provide the basis for a more general discussion of the underlying cognitive processes that appear to be involved in programming activities. In addition, this model serves as an important counterpoint to alternative models of human problem solving that have been concerned more specifically with problem-solving in ill-defined domains and/or with knowledge intensive problem-solving processes.

This chapter then moves on to consider the ACT* model proposed by John Anderson. This model appears to provide a more applicable framework for

understanding problem-solving in programming. Indeed, the model has been applied with some success to analyses of skill acquisition and problem-solving in the context of symbolic programming languages such as LISP. The model of problem-solving proposed by Anderson is important in a number of ways. Firstly, it demonstrates explicitly how skill in a domain can be acquired. Secondly, it shows how both domain specific knowledge and general problem-solving methods can be used to guide problem-solving activities. Finally, the model suggests a set of mechanisms which can be used to explain how problem-solving performance changes with developing expertise. All three of these features are relevant to the issues discussed in this thesis and hence the ACT* model is described here in some detail.

The final part of this chapter attempts to delineate some of the general features of programming tasks that models such as ACT* and GPS do not address. The reason for doing this is that this thesis suggests that the individual features of programming behaviour that have been proposed as component elements in models of problem-solving in programming cannot be considered in isolation. Hence, in order to understand programming behaviour we need to consider the interaction between domain knowledge, features of the device which the programmer uses to create a program and features of the programming language itself which may, in turn, tend to support or to undermine particular forms of programming behaviour. Only a broad characterisation of programming tasks is able to provide a perspective on the complexity of this form of behaviour and this chapter provides a general discussion of those issues which are germane to the characterisation of programming tasks that is suggested by the work reported in this thesis.

2.2 Newell and Simon's General Problem Solver and the problem and state space hypotheses

Simon (1979) presents a generic model of problem-solving activity which suggests a tripartite analysis of problem-solving behaviour. Simon suggests that we consider problem-solving behaviour as an interaction between an information processing system, i.e., the problem-solver, and a task environment, i.e., the task as described by the experimenter. In approaching a

problem-solving task, the problem solver represents the situation as a problem space which includes information on the problem's initial state, its goal state, and the problem-solving operators that may be applied with the intention of reducing the distance between the initial state and the goal state.

Newell and Simon (1972) suggest that when people engage in problem solving behaviour they pass through certain correlative knowledge states. Hence, they begin with a representation of the initial state and search through the space of alternative states until they reach a knowledge state which corresponds to a representation of the goal for the problem. The transformation from one knowledge state to the next is governed by the application of problem solving operators. Since a problem of any complexity is likely to involve a large number of alternative paths, problem solvers can recruit heuristic methods in order to search the problem space more efficiently.

These processes are perhaps best illustrated using an example. The "Tower of Hanoi" puzzle represents the kind of problem to which this model has typically been applied and as such it provides a useful and 'well-constrained' vehicle for demonstrating the application of Newell and Simon's framework. The initial state of the Tower of Hanoi problem specifies a certain configuration of three disks of different diameter placed upon the first of three vertical pegs. Initially, these disks are placed on the peg in size order, the largest at the bottom and the smallest at the top. The goal state for this problem is reached when all the disks are piled in the same order on the last peg.

There are, however, certain restrictions on the way in which disks can be moved, i.e., only one disk can be moved at a time and a larger disk cannot be placed upon a smaller disk. The application of problem-solving operators, which is guided by these constraints, may give rise to a variety of alternative intermediate states between the initial state and the goal state. Since a human problem solver is unable to explore each of these alternatives in parallel, and because exploring every alternative in a serial fashion would be too time consuming for any problem of reasonable complexity, the problem solver needs to rely upon heuristic methods to guide search through the problem space.

One important heuristic method proposed by Newell and Simon is means-end analysis (Newell and Simon, 1972; Newell, Shaw and Simon, 1958). Here the problem solving process procedes by noting the difference between a current state and the goal state, creating a sub-goal which reduces this difference and then selecting and implementing an operator that solves the sub-goal. The means-end heuristic proposes that the problem solver must first evaluate the difference between the current state and the goal state. Secondly, the problem solver must establish a subgoal which reduces this difference and then execute the operator to achieve this subgoal. This process continues until the problem is solved. In contrast to an exhaustive search of the problem space, means-end analysis cannot guarantee a solution although it does reduce the number of alternative states the problem solver has to consider at any one time. Even a simple three ring Tower of Hanoi problem has a total of twenty-seven different states, and the adoption of heuristic search processes such as means-end analysis appears to be one way of obviating the limitations of working memory in problem-solving contexts.

2.2.1 Modelling transformation problems

Polson and his colleagues (Atwood and Polson, 1976; Jeffries, Polson, Razran and Atwood, 1977; Atwood, Masson and Polson, 1980) have conducted a number of studies of transformation problems[1], such as the Tower of Hanoi, which suggest that subjects have a limited understanding of such problems when they are first presented. Polson and his colleagues also suggest that problem solving performance is exacerbated by the inability of problem solvers to plan a sequence of moves if there are more than about three possible successive states.

From their experimental evidence, Polson and his Colleagues have developed a computational model of human problem solving that solves simple transformation problems (Atwood and Polson, 1976). The model incorporates a three-stage process of interactions between means-end analysis and memory processes. The model proposes that information about problem states visited as the problem solver works towards a solution are stored in long-term memory. Since the problem solver is likely to sometimes forget which states they have

visited, the probability that the model will remember a particular state that it has previously encountered is set arbitrarily at .9. Working memory holds information about the current state of the problem, its successor states and the state evaluation information that guides the means-end analysis.

In stage 1, the model selects moves according to the following criteria:

- A move that would lead directly to the goal state is always taken

- A move that leads back to the start state is never taken

- Illegal moves are always rejected

If none of these situations appertain, then the proposed move will be taken if it does not lead to a state that is seen as too far from the goal, and it gives rise to a new state of the problem. In stage 2, the model generates successors in turn and selects the first move that will lead to a new problem state. If none of the successor states leads to a new state, then stage 3 is entered. In stage 3, the number of successor states determines how a move is chosen. If working memory is not overloaded, then the best available move will be taken, i.e., the move that reduces the distance between the current state and the goal state by the greatest amount (that is, the move with the lowest means-end value). If working memory is overloaded (that is, there are more than three successor states), then the model selects a move at random.

This model is important since it supplements the Newell and Simon state-space analysis by providing a full process model of human problem solving for transformational problems. In addition, it specifies the various heuristic methods used by problem solvers and includes assumptions about performance constraints. In general, the model proposed by Atwood and Polson sets forth the following proposals:

- When planning moves, subjects only look ahead to a depth of one move.

- Moves are evaluated by a means-end analysis

- Subjects employ anti-looping heuristics by avoiding moves that return them to immediately preceding states

- There are limitations on the number of successor moves that can be stored and evaluated

This model of problem-solving has some predictive capacity and appears to be generalisable, at least within a broad class of transformational problems that are considered. For instance, one specific prediction that arises from the model is that subjects will only plan one move ahead in order to reduce working memory load. Atwood , Masson and Polson (1980) tested this hypothesis by suggesting that a reduction in memory load will enable the problem solver to plan further ahead. They attempted to reduce working memory load by providing subjects with information about all of the possible moves available from any state in the problem. Atwood et al found that this manipulation produced a small improvement in the subjects' ability to consider alternative moves; however it appears that when capacity is freed it is not used to look ahead. Rather, extra capacity appears to facilitate the avoidance of states that tend to lead back to the initial state of the problem.

The problem solving framework proposed by Newell and Simon and later extended by Polson and his colleagues has proved to be one of the most successful general theories of cognition. The Newell and Simon framework specifies a well articulated process model of the cognitive substrates of human problem solving, and hence it provides an analysis that is amenable to computational modelling. In addition, the model has strong predictive value, and has spawned a significant amount of research concerned with exploring the key predictions of the model.

2.2.2 Problem-solving in ill-structured or ill-defined domains

The success of the Newell and Simon model appears to derive largely from its concern with specifying the role of domain independent and generic problem

solving methods and heuristics. However, it has become clear that there are numerous problem solving domains in which problem solvers need to be able to recruit large amounts of domain specific knowledge in order to guide their problem solving activities. This is especially true of problem domains that might be characterised as ill-defined or ill-structured (Newell, 1969; Reitman, 1965; Simon, 1973; 1978).

Newell (1969) defines ill-structured problems as problems which have poorly defined goals and no well-defined criteria for evaluating the solution to the problem, involving the integration of multiple sources of knowledge, and with no predetermined solution path. In contrast to this, the problems typically addressed by Newell and Simon's model have well defined goal and start states, and the relevant problem solving operators are specified in the problem description. For instance, the legal moves in the Tower of Hanoi problem are derivable from the problem specification, the initial state is given and the goal state is known.

In contrast, it appears that programming tasks display many of the characteristics of ill-structured problems (Guindon, 1988; 1990; Rist, 1989). For example, there are no unique solutions to programming problems - different programs utilising different solution methods may give rise to the same results. As we shall see later in this thesis, programming skill also appears to depend upon the utilisation and integration of multiple sources of knowledge (Brooks 1983; Pennington 1987a and b). The Newell and Simon analysis appears to apply to only a very narrow class of puzzle-like problems and its ability to account for problem-solving in complex real-world domains such as programming is doubtful.

A similar problem is also reflected in VanLehn's (1990) criticism of the problem/state space analysis in terms of its application to subtraction problems. He suggests that the state space hypothesis requires the problem solver to have a test which can inform the search process when the final state or the goal state is achieved. He rules out various possibilities for this test such as asking a teacher or an expert, using superficial approximate tests or adding the answer to the subtrahend to see if it equals the minuend. VanLehn suggests that the last of these tests is implausible since it is infrequently observed, and the second

because it would lead to a much larger class of systematic errors than have actually being found in studies of subtraction. The first test he claims, cannot be used when students are being tested, and that during testing it would be implausible to assume that students search a problem space that is different to the one they searched while learning. Similar criticisms of the problem/state space hypothesis are also evident in the context of programming tasks, since there is not always a clear test that a programmer can apply to inform them whether their program is correct.

In addition to the domain restrictions on the applicability of the Newell and Simon model that are discussed above, there are other reasons for believing that the framework they present is unable to account fully for programming behaviour. In particular, the model proposed by Newell and Simon does not account for the development of problem-solving skill. It might be argued that any model of human problem-solving should provide some account of how problem-solving skills are acquired and how the nature of problem-solving performance changes with the acquisition of these skills. The Newell and Simon framework has been elaborated more recently to include a specific model of learning (Anzai and Simon, 1979), however the emphasis of their model has always been concerned with describing and modelling domain independent problem-solving heuristics, rather than on skill acquisition.

While the Newell and Simon model does not appear to be of immediate relevance to our discussion of problem-solving in programming, it was a major pioneering achievement and remains important for a number of reasons. Firstly, the analysis presented by Newell and Simon proposes a specific nomenclature for describing problem-solving. The terminology they set forth has become almost standard in discussions of problem-solving, and for this reason it has been important to introduce this terminology here. Secondly, our discussion of the Newell and Simon model has required us to distinguish between different classes of problem, i.e., between well-structured and ill-structured problems. This has enabled us to introduce a preliminary characterisation of typical programming tasks and has illustrated that programming should be considered to be a problem-solving activity. Finally, the Newell and Simon model provides an important counterpoint to Anderson's ACT*, which, in contrast to Newell and Simon's approach, places emphasis upon problem-solving in knowledge

intensive domains and suggests mechanisms which may underpin the development of problem-solving skill.

2.3 Anderson's ACT* model of problem-solving

Anderson's (1983) ACT* (Adaptive Control of Thought) framework specifies a generic cognitive architecture which has been used to account for a range of cognitive functions, from pattern recognition to problem-solving. In addition, it has provided a basis for theories of skill acquisition in a number of domains including text editing (Singley and Anderson, 1985; 1988) , geometry (Anderson, Greeno, Kline and Neves, 1981) and computer programming (Anderson and Reisner, 1985; Pirolli and Anderson, 1985; Pirolli and Bielaczyc, 1989). Anderson's framework provides a detailed description of the processes which govern skill acquisition and which lead to the performance differences that have typically been associated with skill development. Since a primary aim of this thesis is to attempt to specify the processes that give rise to the performance differences that are observed to accompany the development of programming expertise, a discussion of the ACT* framework is of particular relevance.

ACT* has three main structural components; a declarative memory, a production (or procedural) memory and a working memory. Figure 2.1 is a schematic representation of the ACT* architecture showing its major structural components and their interlinking processes.

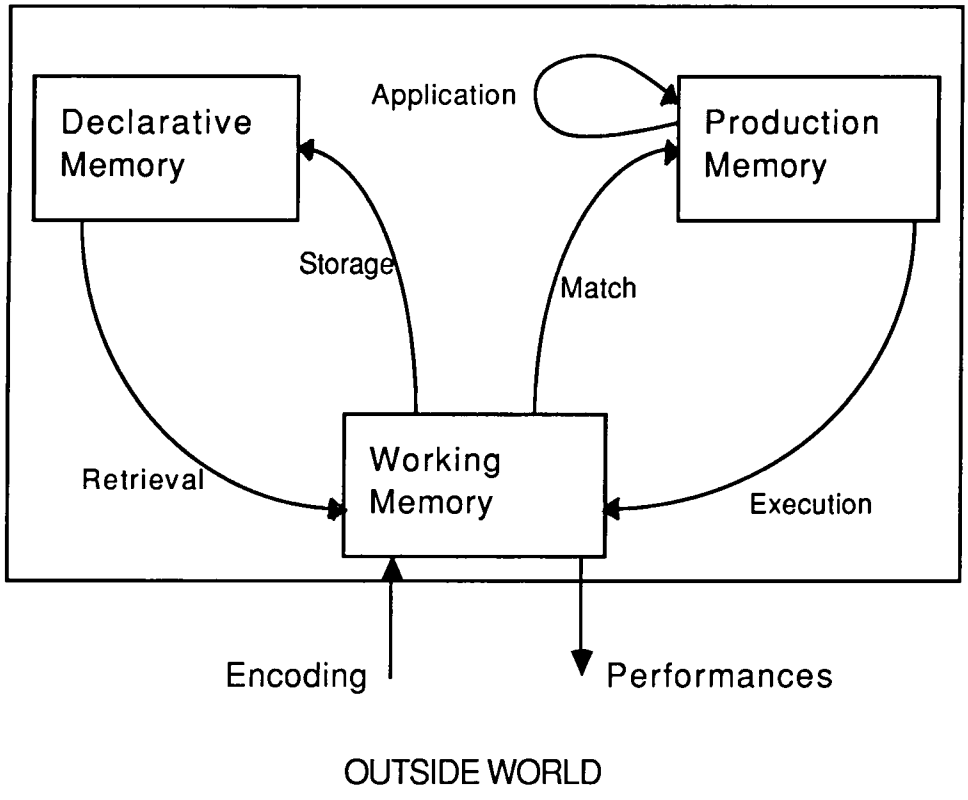


Figure 2.1 A schematic representation of the ACT framework*

The declarative memory of ACT* is represented as a semantic network (Collins and Loftus, 1975) of interconnected nodes which have different activation strengths. These nodes represent cognitive units or chunks which can be such things as propositions (hate, Bill, Fred) strings (one, two, three) or spatial images (a triangle above a square). Each of these cognitive units encodes a set of elements in a particular relationship. Anderson restricts the size of these cognitive units to five elements, and more complex structures are formed by constructing a hierarchical network of simple elements.

The procedural memory of ACT* is basically a production system (Hunt and Poltrack, 1974, Young 1979). Production systems consist of a collection of IF (state) - THEN (action) rules, which specify various condition-action pairs and some global procedure for instantiating rules and for performing a consequent action. The condition of the production rule specifies some data pattern, and if

elements matching these patterns are found in working memory, then the production will 'fire' - i.e., operate. The action part of the rule specifies what to do in that particular state. A typical production rule, informally stated, might take the following form:

IF person 1 is the father of person 2
    and person 2 is the father of person 3
THEN person 1 is the grandfather of person 3.

This production would apply if 'Fred is the father if Bill' and 'Bill is the father of Tom' were active in working memory. This rule would enable the system to infer that Fred is the grandfather of Tom and would deposit this fact in working memory. Since production systems are computationally universal[2] they are able to model any class of cognitive activity that we are able to specify. Indeed, production system models have been used by various researchers to explain a wide range of cognitive processes (for instance, see Brown and Van Lehn, 1980; Kieras and Bovair, 1981).

The third main structural component in Anderson's model is working memory. In fact, working memory is not a separable component in this model, but rather it represents that information in declarative memory that is currently active. According to Anderson (1983) "Working memory consists of information that the system can currently access, consisting of information retrieved from long-term declarative memory as well as temporary structures deposited by encoding processes and the action of productions" (p. 19).

Most of the processes shown in figure 2.1 involve working memory. Encoding processes deposit information derived from the outside world into working memory, while performance processes convert commands contained in working memory into action. Anderson claims that, unlike the other processes specified by ACT*, these two processes are not central to the framework. The storage process basically modifies the contents of declarative memory by either creating a permanent record in declarative memory of the contents of working memory or by altering the strengths of existing records in the declarative system. The

retrieval process retrieves information from declarative memory. The storage and retrieval of information in declarative memory can be effected in a number of ways. For instance, when the rules in the production memory match the contents of working memory they are executed. The process of production matching, followed by execution that is specified by these processes is called production application. Production memory can also be applied recursively, such that new productions can be formed by examining existing productions.

Anderson is careful to suggest that this general framework does not constitute a theory, since it makes no specific predictions about behaviour. Anderson suggests that a predictive theory must specify in greater detail the properties of the storage, retrieval and production application processes that are central to the ACT* framework. In fact most of Anderson's book 'The architecture of cognition' is devoted to a consideration of these issues, and it is not feasible to review them here in any detail. However, one important aspect of the ACT* framework is its ability to account for skill acquisition, and it is to this we now turn. In addition to outlining the general theoretical principles of skill acquisition in ACT*, the next section of this chapter shows how ACT* has been applied to skill acquisition in a programming context.

2.3.1 ACT* and skill acquisition

A central tenet of the ACT* theory is that skill learning consists largely of a process which transforms declarative knowledge into procedural knowledge and then modifies this procedural knowledge via application processes. This declarative/procedural knowledge distinction is clearly one of the fundamental elements of Anderson's theory of skill acquisition. Broadly speaking, declarative knowledge is knowledge that is open to introspection and can therefore be reported and is not tied to the situation in which it is used. In contrast, procedural knowledge is applied automatically, often cannot be reported and can only be applied in specific situations.

Stated broadly, Anderson views skill acquisition as a move from the use of declarative knowledge structures to procedurally based knowledge that can be applied rapidly and automatically in specific situations. More specifically,

Anderson claims that the acquisition of cognitive skills consists of three successive stages of learning; a declarative stage, a procedural stage and a tuning stage.

The first stage of skill acquisition involves the accumulation of domain relevant facts which are incorporated into declarative network structures. This knowledge will be used in conjunction with domain independent heuristics, such as means-end analysis. For instance, when learning chess, the novice would acquire a number of rules or facts which characterise the legal moves for each piece. Similarly, in learning programming, the novice programmer may rely exclusively upon the programming knowledge they have acquired from textbooks. One feature of declarative knowledge is that it does not require the problem-solver to use it in some specific manner, as would be necessary if that knowledge were represented in a procedural fashion. However, before declarative knowledge can be used it must be retrieved and kept active in working memory. The ACT* framework suggests that the slow pace and tentative nature of problem-solving during this declarative stage can be attributed to the need to activate and access information in long-term memory (Anderson, 1982). Moreover, the loss of information from working memory can account for many of the errors made by novice problem solvers (Anderson and Jeffries, 1985).

As problem solvers become more experienced, Anderson claims that declarative knowledge becomes proceduralized. During this so called transitional stage, productions are created from the declarative knowledge acquired during the first phase of skill acquisition. In this second stage, successful sequences of activity, produced by the application of weak method heuristics to declarative knowledge are complied into new domain-specific productions. This compilation process governs the transformation from the interpretive application of declarative knowledge to procedures that apply this knowledge directly (Neves and Anderson, 1981). Knowledge compilation has adaptive value in the sense that it eliminates the retrieval process, and by doing so it has the effect of not only speeding up performance but also reducing the load on working memory.

The knowledge compilation processes in ACT* take two distinct forms: composition and proceduralization. Composition (Anderson, 1976; Lewis,

1978) takes a sequence of productions that follow each other in solving a particular problem and collapses them into a single production that has the same effect as that sequence. Proceduralization (Anderson, 1982) takes place when a particular piece of declarative knowledge is used repeatedly in the context of a particular subgoal and results in the creation of a production rule which represents the declarative information as its condition and the result of the execution of this declarative information as its action.

In the final stage of skill acquisition, any additional learning is attributed to the tuning of procedures. Basically, this tuning process involves an improvement in the choice of procedures for performing a given task. Since one can characterise all problem-solving tasks as involving search through a problem space, then the tuning of this search procedure, such that it leads to the discovery of optimal solutions, will presumabley improve problem solving performance. Anderson, Kline and Beasley (1980) have proposed three learning mechanisms which may be employed in this search tuning process. They suggest a generalisation process by which production rules become broader in their range of applicability, a discrimination process in which their range is narrowed, and a strengthening process by which successful rules are strengthened and poorer rules weakened.

2.3.2 Applying the ACT* framework - Skill acquisition in LISP

Anderson, Conrad and Corbett (1989) describe an analysis of student learning with their LISP tutor which adopts the general principles of the ACT* framework. Anderson et al suggest that there are basically three claims that can be derived from the ACT* model that are relevant to the instruction of a skill such as LISP. According to the ACT* theory a skill like LISP programming can be represented as a set of production rules. For instance, the following piece of code, which implements a function that inserts the second element of one list at the beginning of another list, can be represented as a series of productions:

Hence the code;

```
(defun insert-second (lis 1 lis2)
        (cons (car (cdr lis 1)) lis2))
```

becomes the production rule set;

p-defun

IF the goal is to define a function
THEN code defun and set subgoals
1. To code the name of the function.
2. To code the parameters of the function
3. To code the relation calculated by the function

p-name

IF the goal is to code the name of the function
and = name is the name
THEN code = name

p-params

IF the goal is to code the parameters of the function
and the function accepts one or more arguments
THEN create a variable for each member of the set
and code them as a list within parentheses

p-insert

IF the goal is to get the second element of a list
THEN code cons and set subgoals
1. To code the element
2. To code the list

p-second

IF the goal is to get the second element of a list
THEN code car and set a subgoal
1. To code the tail of a list

p-tail

IF the goal is to code the tail of a list
THEN code cdr and set a subgoal
1. To code the list

p-var

IF the goal is to code an expression
and a function parameter has the expression as a
value
and = name is the name assigned to that
parameter
THEN code = name

Anderson et al have derived about 500 production rules of this kind to encode the skill of programming in LISP. Although the ACT* theory holds that the knowledge of a skilled programmer will be represented in this way, this is not the case for a novice programmer. Anderson et al suggest that one cannot present these rules to a student and expect the student to directly encode them as production rules. Rather, information must initially be encoded in declarative form. Anderson et al, on the basis of informal observation, suggest that during learning, students rely to a greater extent upon exemplar descriptions of the functions of LISP constructs rather than upon rule-based descriptions. Some empirical support for the efficacy of exemplar-based approaches to teaching programming is presented by Boyle and Drazkowski (1989).

Once the student has acquired some declarative knowledge of the domain, the knowledge compilation process converts the initial interpretative use of this declarative knowledge into a procedural production rule form. Anderson, Boyle, Farrell and Reisner (1987) provide an example of this compilation mechanism in the context of skill development in LISP programming. In their model there is a function that will retrieve function definitions from long-term memory and apply them in appropriate contexts. For instance, in the following production, relation and function are variables which allow the production to match different data:

IF	the goal is to code a relation defined on an argument and there is a LISP function that codes this relation
THEN	use this function with the argument and set a subgoal to code the argument

Here, the second line of the condition might match, for instance, 'CAR codes the first member of the list'. This rule can be proceduralized to eliminate the retrieval of the CAR definition as follows:

IF	the goal is to code the first member of a list
THEN	use the CAR of the list and set as a subgoal to code the list

This proceduralization is achieved by deleting the second clause that required long-term memory retrieval from the first production. Moreover, the rest of the production is made specific to the relationship between *first member* and the function CAR. Once a production has been created that can directly recognise the application of CAR, this will result in a reduction in the quantity of long-term memory information that needs to be held in working memory.

Another compilation mechanism that is important in the ACT* framework is composition. Basically, composition involves combining a number of successful operators into a single macro-operator that has the same overall effect as the sequence of individual operators. Anderson et al again provide an example of this mechanism from LISP. Suppose that one wanted to insert the first member of list1 into list2. Here the following two operators would apply in sequence:

IF	the goal is to insert an element into a list
THEN	CONS the element to the list and set as subgoals to code the element and to code the list
IF	the goal is code the first member of a list
THEN	take the CAR of the list and set as a subgoal to code the list

Here the first rule would apply and bind *an element* to 'the first member of list1' and a *list* to 'list2'. The second production would apply and bind a list to 'list1'. A simple case of composition involves collapsing these two productions into a single production:

IF	the goal is to insert the first member of one list into another list
THEN	CONS the CAR of the first list to the second list and set as subgoals to code the first list and code the second list

The result of composing productions in this way would be an increase in coding speed since a problem could be coded in fewer steps. McKendree and Anderson (1987) have provided some empirical support for this speedup phenomenon for frequently repeated combinations of LISP functions.

Anderson et al (1989) present a number of empirical studies of their LISP tutor. This tutor embodies several design considerations derived from the ACT* model of skill acquisition. While it is not appropriate to discuss the design of this tutor here, the empirical findings of Anderson et al (1989) are of relevance to the current discussion, since they provide some support for their model of skill acquisition in LISP.

In one of their analyses, Anderson et al looked for learning trends amongst their subjects who were using the LISP tutor. They found evidence to support the view that production rules are modular units of knowledge which can be learned independently from other units. Hence, subjects show regular learning curves defined on production rules and their independence from other similar rules was demonstrated by subjecting the production learning data to factor analysis. They also found that these rules are abstract and are not tied to a specific content. Hence, production rules can be transferred across contents and languages. They suggest that these findings not only provide support for the ACT* model but are also contrary to other theories of skill acquisition, such as those proposed by schema theory (Rumelhart, 1980; Rumelhart and Norman, 1981) and by connectionist approaches (McClelland and Rumelhart, 1986; Rumelhart and McClelland, 1986). For instance, they claim that schema theory would emphasise larger units of knowledge than productions and that connectionist approaches would rule out the existence of abstract rules such as those evident in the Anderson et al study.

The ACT* framework has provided a detailed specification of skill acquisition in a number of domains and Anderson and his colleagues have constructed computational models of skill acquisition which embody the main assumptions of the ACT* framework. These models have demonstrated that a learning system which starts with only a small number of domain related facts stored in declarative memory, together with a number of preexisting problem solving procedures can acquire new procedures which can be made responsive to the kinds of situations that occur in the domain of learning. ACT* suggests a basic set of learning processes and specifies, in effect, a theory of the basic principles of operation built into the cognitive system. The main strength of the ACT* framework is that it can explain a range of relevant phenomena from the performance differences associated with developing expertise to the transfer of cognitive skill.

2.3.3 Problems and limitations of the ACT* framework

In one sense the main problem with ACT* is that it is too powerful. For instance, many of the simulations of skill learning which embody ACT* principles often perform better than the human subjects that they are intended to model. For example Anderson (1982) describes a simulation of the behaviour of high-school students solving two-column proof problems in geometry. While the simulation managed to solve the problems that it was presented with, not all the students did. This suggests that ACT* may provide a good model of idealised problem solving and learning, but not necessarily of the actual problem solving behaviour exhibited by real subjects.

This problem is compounded to some extent by the focus of the ACT* framework upon models of appropriate skilled performance rather than upon mistaken and/or inappropriate behaviour. The ACT* framework suggests that errors in problem solving are either 'systematic' errors derived from defective knowledge or are errors which arise from unintended actions or slips (Norman, 1981; Reason, 1979) during procedure execution . However as Brown, Burton and VanLehn have shown, errors can arise because individuals may have 'buggy' procedures, i.e., correct procedures with one or more minor perturbations or bugs (Brown and Burton, 1978; Brown and VanLehn, 1980; Burton, 1982; VanLehn, 1982, 1990). Anderson et al (1987) have attempted to build buggy procedures into their LISP tutor and these procedures are used to account for student errors. However, even with the inclusion of buggy procedures, it is still possible that there are other mechanisms which can produce regular errors.

For instance, it is possible that a problem solver may simply have no idea how to solve a particular problem and hence may adopt a coping strategy to produce a response. This problem suggests that we may require richer representations of skill than are presently afforded by models such as ACT*. Experts in some domain can readily cope with novel problems; are often able to transfer their skills to very different domains and can argue and reason about the subject matter of their domain and the nature of their expertise. As we shall see later, in the programming domain, these skills are clearly important. However, the ACT* model addresses none of these abilities, since it concentrates upon a parsimonious explanation of performance in the context of well practised and familiar tasks.

Payne (1988) suggests that frameworks such as ACT* fail to consider the importance of non-procedural conceptual representations in problem solving and that "even skills with a large procedural component rely on a great deal of conceptual knowledge, and that the content and structure of the conceptual knowledge may continue to develop with expertise, alongside the procedural methods" (pg 76). However, as we have seen, the ACT* framework suggests that non-procedural knowledge ceases to play a role in skill after the initial novice stage. In chapter 3 a number of studies are reviewed which place a strong emphasis upon the role of conceptual knowledge in programming expertise. It appears that in programming, as well as in other domains, non-procedural knowledge plays a vital role in expertise, and that the ACT* framework is limited it terms of its ability to account for expertise in programming or similar knowledge intensive domains.

2.4 Can existing models of problem-solving account for the complexity of programming behaviour?

The two problem solving frameworks that have been reviewed in this chapter appear to display a number of general limitations which suggest that they may be unable to account for the complexity evident in programming tasks. For instance, we have characterised programming problems as ill-structured tasks which are highly knowledge intensive. The state space analysis of problem solving suggested by Newell and Simon would appear to be unable to account for problem solving behaviour in ill-structured domains, and as such it provides an inappropriate model for attempting to understand the full range of problem-solving activities involved in programming.

The ACT* model is advanced as an alternative framework for understanding problem solving and skill acquisition in knowledge intensive domains. Consequently, this model should provide a more appropriate framework for understanding programming behaviour. However, as we have seen, the ACT* framework fails to suggest a role for non-procedural or conceptual knowledge in problem solving. Since many studies of programming highlight the importance of such knowledge (see review in chapter 3) in the development of expertise, it may be doubted whether models such as ACT* provide sufficient flexibility to

characterise the many facets of expertise which have been demonstrated in a range of studies.

The limitations of the two frameworks that have been described above not only present problems in terms of an analysis of programming behaviour, since they are clearly of significance in other problem solving domains. However, it is apparent that there are certain salient properties of programming tasks which appear to be very specific to the problem solving activities that occur in this domain. Moreover, extant problem solving frameworks do not appear to provide a basis that can fully account for the effects that these features of programming tasks can have on problem solving behaviour. The next section of this chapter concentrates upon a discussion of these issues. This provides a basis for a characterisation of programming activates which not only considers the nature of the programmer's task, but in addition, highlights the role in problem solving of programming language features and of the tools used by programmers to create programs. This multifaceted view of problem solving in programming reflects the broad approach adopted by this thesis.

2.4.1 The inseparability of planning and execution

The dominant view of problem-solving suggests that the problem-solving activity is a top-down focused process that starts with high-level goals which are subsequently refined into subgoals and ultimately into achievable action. This process is often referred to as successive refinement or problem decomposition. Recall that the model of problem-solving suggested by Newell and Simon starts with a specification of the goal state and attempts to reduce the difference between the initial or the current state of the problem by applying operators or by creating subgoals. In routine skill, this decomposition process can take place without problem solving since action sequences can be remembered as prepackaged methods for achieving specific goals (Newell and Simon, 1972; Card, Moran and Newell, 1983). In this situation information from the external world plays a limited role by providing feedback information to test the effects of certain operators against goals (Miller, Galanter and Pribram,1960; Norman, 1986).

This account of human problem solving has clearly been successful. However it under emphasises the role of perception in cognitive skill and the extent to which the external world can act as a resource to support problem solving behaviour. A number of researches have shown that the 'external display' upon which a task is enacted can serve as a repository for search control knowledge and can therefore reduce the working memory load that is normally required to carry out a given task. One can compare the use of external displays with classical models of problem solving where the problem solver must maintain a stack of subgoals. The importance of external memory props has been demonstrated by several researchers who have suggested that actions are often enacted before plans are complete (for example see, Green, Bellamy and Parker, 1987; Young and Simon, 1987). In addition, an important parallel development in AI models of planning is the idea that planning cannot be easily separated from execution (Agre and Chapman, 1987; Ambros-Ingerson, 1987; Chapman, 1987; 1989). For instance, Ambros-Ingerson (1987) highlights the importance of 'knowledge getting actions', which are built into plans and specify where additional planning information can be obtained, generally from external sources. Larkin's (1989) recent analysis of 'display-based' problem solving similarly stresses the importance of obtaining information about the current problem state from the display.

In the programming domain, Green et al (1987) have proposed a model of programming behaviour - their so called 'Parsing/Gnisrap model - which suggests that because of working memory limitations, programmers must dump information onto an external medium (i.e., a VDU screen) when overload is threatened and then later parse that information back into a cognitive representation when it is subsequently required. The Parsing/Gnisrap model is described in greater detail in chapter 4, but one important contribution of this model is the idea that planning and execution in programming are inextricably bound together. The existence of such a phenomenon poses problems for classical models of problem solving in terms of their ability to account for problem solving behaviour in programming.There are two main reasons for this. Firstly, such models fail to recognise the importance of external media in problem solving and secondly, they suggest that planning and plan execution (or operator selection and execution) are separate processes, whereas, in fact, in many complex domains, they appear to be very closely linked.

2.4.2 The effects of programming language features on programming behaviour

Programming tasks are clearly rather different from the kinds of tasks that are typically studied in problem solving analyses. One major difference is that programming involves the use of a specialised language which is employed to implement the various problem solving operators necessary to solve a particular problem. In traditional problem solving studies, operators are applied directly to objects in the world, and it is not necessary to employ a metalanguage to describe or to express these operators. One can of course describe games such as chess using a formal notation. However, it is likely that the various problem solving strategies or methods which are employed while playing a game of chess may well be different to the methods and strategies used if one has to describe and 'play' the game using an abstract formal notation[3].

While we might attempt to characterise programming languages as in some sense analogous to the formal language that can be used to describe chess moves, there does not appear to be a close programming analogy with the more typical situation where problem solving operators are applied directly to objects in the world. Some writers have claimed that one way of making programming more 'natural' might be to use graphical notations, in which graphical objects representing various programming constructs can be manipulated in various ways to construct a program (Badre and Allen, 1989; Cunniff and Taylor, 1987a and b; Cunniff, Taylor and Black, 1986). Others have suggested that natural language might provide a better means of specifying algorithms (Biermann, Ballard and Sigmon, 1983). There is a great deal of current debate surrounding the use and the general efficacy of these various notations (Dyck and Mayer, 1985; Fitter and Green, 1979; Green, 1982; Galotti and Ganong, 1985). However what is fairly clear is that different notations may support different kinds of problem solving tasks to a greater or a lesser extent.

For instance, Adelson (1984, and see review in chapter 3) has shown that the way in which programming information is presented (i.e., as different types of flowcharts or as text) can have a marked effect upon subjects' ability to answer different kinds of program comprehension question. Hence, it appears that subjects may be able to extract different forms of information more readily from different notations. This phenomenon does not only arise when comparing text

with graphical presentation, since other studies have demonstrated that different text-based notations can support different kinds of programming tasks.

For example, Green, Bellamy and Parker (1987) have shown that programmers rarely generate programs in a strictly linear fashion, where the final text order of the program mirrors its generative order. Moreover, this study demonstrated that the extent of this non-linearity is related in part to the programming language that subjects use. Green et al claim that some notations facilitate linear generation while others do not. Additional support for this is provided by Bellamy and Gilmore (1990) who found that Pascal programmers adopted a non-linear style of program development, while BASIC and PROLOG programmers exhibited a more or less linear style . They claim that "the task of translating from the plan structure to the linear structure of Pascal places far too high a mental load even on expert programmers, doing simple problems. In other words, the Pascal programming language fails to support plan generation" (pg 68). They go on to suggest that programming plans (that is plans that represent stereotypical tacit programming knowledge) "are dependent upon the programming strategy of the particular programmer. This in turn is influenced by the notation, taught strategies and the programming environment used during learning to program" (pg 68). Hence, it appears that certain notational properties of programming languages can affect the problem solving strategies exhibited during program generation.

In addition, language features also appear to affect the comprehension of programs. For example, Gilmore and Green (1988) have shown that programmers can extract certain forms of information from some notations more easily than they can from others. They introduce the term "role-expessiveness" to describe the extent to which certain notations may make particular programming languages more discriminable from others. They suggest that one primary element of role-expressiveness relates to the ease with which programmers are able to extract certain forms of information from a given program text. They claim that this process will be facilitated if the programmer is able to discriminate between the structure she is searching for and other structures that surround it. For example, Gilmore and Green claim that Pascal is more role-expressive than BASIC since Pascal has a rich set of lexical cues, making structure more discriminable, and in BASIC the same piece of code may be used for more than one purpose, which can lead to structure confusion. Role expressiveness is clearly not the only notation

feature that might contribute to particular kinds of programming strategy. In fact, Green (1989) has outlined a number of so called notational dimensions, which he claims may determine the strategy employed by programmers when engaged in particular tasks.

Hence, another problem with classical accounts of problem solving behaviour is that such accounts fail to generalise to problem domains in which problem solving operators need to be expressed in some metalanguage. As a consequence, such models cannot account fully for the problem solving behaviour exhibited by programmers which appears to be heavily influenced by certain properties of the notation of the particular language being used.

2.4.3 The effects of problem solving environments on behaviour

There appear to be two major cognitive demands that distinguish computer use, including normal programming tasks, from the puzzle solving concerns traditionally addressed in studies of problem solving (Payne; 1987; 1990). Firstly, when using a computer system, operators cannot be applied directly. Rather they must be effected via a task language which maps operators into action sequences. Secondly, the structure of the problem space is clearly more complex, since in addition to representing goals and subgoals in the task domain, it must also accommodate some representation of the device. In programming terms we might the distinction between the 'device language', which the programmer uses to issue commands to the editor or its equivalent and the 'task language' which represents rules that describe the target domain (Green, Bellamy and Parker, 1987). In addition, we need to consider the medium by which the user interacts with a task (i.e., the interaction medium), e.g., pen and paper, VDU etc. The characteristics of each of these elements of interactive behaviour are likely to play a role in the determination of programming behaviour or strategy.

This is amply illustrated in Green's discussion of a system designed to receive spoken Pascal code, using an isolated-utterance speech recognizer (Green, 1989). Green claims that "the system worked reasonably well - i.e., if one dictated Pascal code into it, the recognition rate was quite acceptable. However, because the speech recogniser relied upon the constraints of Pascal to make the recognition

problem tractable, the program had to be syntactically correct at all times. The design of the system made it preferable to dictate the program in text order, from start to finish. Unfortunately, because of certain characteristics of the Pascal notation, it is pretty well impossible to dictate impromptu Pascal without any omissions...Pascal is not a suitable notation for this environment" (pg. 444).

Green, Bellamy and Parker (1987) have demonstrated that programmers rarely generate their programs in a strict linear order. Hence, one might hypothesise that if programmers are constrained to a linear generation strategy, then this will seriously affect their performance. In chapter 11 an experiment is reported in which programmers were required to use an editor which did not allow any backward movement. Hence, like the speech input system discussed by Green, programmers were forced into a situation where they had to generate their programs in a linear fashion. Unlike the speech input system, this editor displayed the program generated to date, and hence reduced the demand on working memory to some extent. However, programmers still made many errors. Indeed, when programmers had to use this environment, the performance of expert programmers was reduced to the level exhibited by novices. This decrement in performance serves to illustrate the effect that device characteristics and interaction medium can have on performance, and suggests a major limitation of traditional problem solving frameworks which fail to consider the effects of the device and interaction medium on behaviour.

2.5 Conclusions

This chapter has been concerned with a review of two important problem solving frameworks and has placed particular emphasis upon the ability of these frameworks to account for problem solving behaviour in programming. While neither of these frameworks appears to provide an account of programming behaviour that can capture its full richness, they do at least provide a starting point from which one can attempt to begin to characterise programming behaviour. This chapter has suggested that we cannot view programming as an activity which simply involves the application of generic problem solving methods and nor can we model its full complexity by simply suggesting a large collection of condition-action rules and some means of applying them. Rather, programming

behaviour appears to depend upon the use of complex planning and problem solving strategies which in turn may be supported or otherwise by certain notational features of programming languages and by features of the device space. As Bellamy and Gilmore (1990) suggest:

" To understand the psychology of complex tasks such as programming, we need to consider planning strategies in particular task contexts. Theories of planning and problem solving have spent too long with their heads in the sand, ignoring the role the external world plays in determining behaviour. If psychology is going to make significant contributions both in theoretical and applied areas of research, we need investigations of how features of the external world determine behavioral strategies. Only then will we be able to produce artifacts that support effective task strategies" (pg 69-70).

The work reported in this thesis broadly supports this view, but perhaps more importantly, it suggests that we need to consider not only the role of the external world in the determination of programming behaviour, but also the way in which representations of programming knowledge interact with language features to give rise to particular forms of strategy. This tripartite view of programming behaviour is reflected in the concerns of subsequent chapters, where these issues will receive further attention. In chapter 3, we review the extensive literature on knowledge representation in programming, while chapter 4 concentrates upon the nature of the problem solving strategies which appear to underpin programming behaviour. In chapter 5, concern is directed towards a review of studies which have suggested that programming language features play a major role in the determination of programming strategy. This provides a basis for the experiments reported in chapters 6 - 11, which in turn provide support for the tripartite analysis of programming behaviour advanced in this thesis.

Notes

[1] Greeno (1978) introduces the term 'transformation problem' to characterise those problems which transform one situation into another. Hence, the Tower of Hanoi problem is transformational in that one can identify various moves which transform one state into another. Other examples of transformational problems are the water jugs problem (Atwood and Polson, 1976) and the missionaries and cannibals problem (Simon and Reed, 1976)

[2] Production systems are computationally universal in the sense that they can be implemented as a Turing Machine and according to the Church-Turing thesis, any behaviour that can be precisely specified will be in the class of things that can be computed by a Turing Machine.

[3] For instance one cannot easily extract perceptual groupings from a formal symbolic representation of chess positions and moves and the ability to do this would appear to be a central element of skill in this domain.

Chapter 3. Knowledge Representation and Expert/Novice Differences in Programming and in Related Domains

3.1 Introduction

A large number of studies concerned with the nature and development of expertise in programming and in other problem solving domains have highlighted significant qualitative differences between the way in which experts and novices organise and structure their knowledge about a particular domain. Studies of expert/novice differences in programming have drawn extensively from theories of expertise in other domains. For instance, a number of studies, following de Groot's (1965) seminal work on expertise in chess have attempted to relate skill differences in programming to the ability of experts to recognise and represent meaningful chunks of code (Barfield, 1986; Shneiderman, 1976). Other theories of programming skill have adopted generalised production system architectures to explain certain salient attributes of expert performance, including phenomena that are typically associated with skill development such as knowledge restructuring and compilation (see chapter 2).

A number of other important studies have drawn analogies with work in the text comprehension domain (Détienne and Soloway, 1990; Soloway and Ehrlich, 1984). Studies of knowledge representation in text comprehension have been germane to both the development of theoretical accounts of the content of tacit programming knowledge and to descriptions of the organisation and structure of this knowledge. More recently the importance of hybrid representations has been highlighted. This hybrid approach suggests that programmers can recruit knowledge from a variety of diverse sources in order to guide problem-solving and, in addition, that a simple uniform view of knowledge representation in programming is not sufficient to account for the behavioral complexity of this task.

This chapter will provide a thematic review of existing studies which have been concerned with the relationship between knowledge structure and expertise in programming. Existing work on knowledge representation in non-programming domains will not be comprehensively reviewed since many studies of expertise in programming have served to replicate the basic findings of research in other domains. Hence, the main aim of the present chapter is to provide a broad review

of existing research into knowledge representation and skill differences in programming. However, this focus on programming research provides a cogent and extensive overview of some of the more general findings to emerge from the problem-solving literature concerned with domains such as Chess, Physics and Mathematics.

This does not mean to say that work in such domains has nothing more to contribute to a study of problem-solving and skill differences in programming. Rather, it will be suggested that a great deal of theoretical understanding still remains to be gained from an analysis of problem-solving in these kinds of domain. Moreover, it should be noted that while many early programming studies appear to draw upon previous problem-solving research only in as much as they attempt to replicate its basic findings, more recent work is beginning to contribute more explicitly to our theoretical understanding of general problem-solving skills in addition to providing a description of the content and structure of knowledge representation for programming tasks.

This chapter begins by reviewing some of the early work on skill differences and knowledge representation in programming. These studies serve to provide important supplementary support for the now well known finding that experts tend to represent domain knowledge in terms of meaningful chunks, while novices tend to derive cognitive structures from salient surface features of problems. More recent work has elaborated in greater detail the nature of these different grouping strategies, while studies of programming that claim allegiance to the text comprehension paradigm have specified the content of the stereotypical knowledge structures that are brought to bear during problem-solving.

This chapter concludes with an evaluation of these studies in terms of their contribution to our understanding of problem-solving in programming. It will be suggested that many of these studies have failed to elicit information about cognitive structure and content that can be generalised between programming languages and paradigms. The main criticism of such work is that theories and principles have often been derived from studies of the comprehension of a single programming language and that the implications drawn from such studies do not necessarily apply in the case of different languages or different programming paradigms.

Another aim of this chapter is to provide an overview and background to a number of the empirical studies reported in this thesis which have addressed issues relating to the generalisability of existing studies. This empirical work will not be discussed in detail in the present chapter but will be briefly mentioned in the final section in order to place it in the context of other work and to illustrate its historical anticedents and development. This chapter attempts to draw out links between the individual experimental studies reported in this thesis, and concludes by suggesting that a more comprehensive understanding of the cognitive aspects of programming will arise only when programming skill is interpreted in a broad ecumenical context. That is, in a context which empahsises not only aspects of a programmer's knowledge representation but also considers the development of the strategic aspects of programming skill and the way in which the notational properties of certain languages might act to facilitate or to constrain programming behaviour.

3.2 Expert/Novice differences and chunking skills

The idea that experts develop 'chunks' of knowledge that represent important functional units or structures within a particular domain is common psychological currency (Miller, 1956; Chase and Simon, 1973; de Groot, 1965; Egan and Schwartz, 1979; Larkin, McDermott, Simon and Simon, 1980; Reitman, 1976). Within the programming domain, a range of studies have suggested that programming expertise might be characterised by the programmer's ability to decompose and represent programs in terms of 'chunks' of knowledge which are based upon semantically meaningful elements of programs. For example, Shneiderman (1976) has shown that the standard results obtained in the now well known chess studies are replicable in the context of programming. Shneiderman found that experts could recall more lines of program code than novice programmers when that program was organised in executable order. Conversely, when the program was ordered randomly no significant differences in recall performance between the two groups were evident.

A number of other studies, employing more complex recall procedures and stimulus material, have provided additional support for the 'chunking' hypothesis (Bateson, Alexander and Murphy, 1987; Barfield, 1986). For instance, Barfield (1986), employed three levels of program organisation (executable order, random lines and random chunks) in a free recall experiment to explore differences in

knowledge representation for programmers of different skill levels. Barfield found that novice programmers recalled the same number of program lines regardless of program organisation. Conversely, the intermediate group demonstrated superior recall when the program was presented in executable order as opposed to the random line and random chunk conditions. Barfield suggests that this indicates that executable order facilitates chunking for intermediate level programmers. Experts, on the other hand, demonstrated high recall performance in both the executable order condition and in the random chunk condition. However, when the program is completely randomised the performance of the expert group drops to a similar level to that exhibited by intermediates. Barfield suggests that since the memory capacity of the experts is the same as that for other groups, the superior recall in the random chunk and executable order conditions must arise because of the experts familiarity with particular units of code that perform the same function.

3.3 Knowledge structures: content and formation

The studies reviewed above demonstrate that the superior organisation of programming knowledge in long term memory is one of the central factors in expert performance. However, these studies fail to specify in detail the nature of that organisation. For instance, one question that is left unresolved relates to the content of expert knowledge structures. In addition, the chunking studies described above do not address issues concerned with the mechanisms which may underlie an expert's ability to create and represent meaningful chunks of knowledge. As a consequence, two central questions appear to emerge from these studies; How are chunks formed and what sort of information do such structures typically represent ?

3.3.1 The organisation of programming knowledge by novices and experts

McKeithen, Reitman, Rueter and Hirtle, (1981) describe a study which attempts to address the issues raised by these questions. They report two experiments. The first experiment replicates the classic expert-novice difference in short term recall for programmers who viewed either a coherent or a scrambled version of a program. This experiment produced results which parallel those of Barfield (1986) and of Sheiderman (1976) reported above. In a second experiment, McKeithen et al., attempted to extend the interpretation of this basic phenomenon by inferring

details of a subject's organisation of programming concepts by constructing hierarchical representations of the relations among programming language keywords.

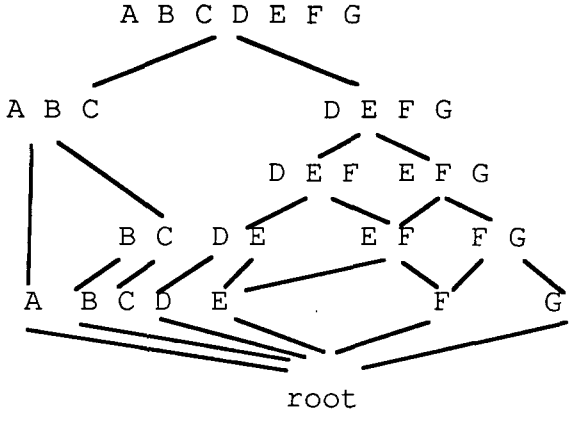
For this experiment, McKeithen et al. employed a technique developed by Reitman and Rueter (1980) which produces a hierarchical representation of information from regularities in the orders in which items are recalled over many trials. This technique capitalises upon the fact that people tend to recall all items of one chunk before moving on to the next chunk (Cohen, 1966). McKeithen et al. examined the way in which programmers form chunks by presenting their subjects with a number of cards each of which had written upon it a word derived from the set of ALGOL W reserved words (for instance, ELSE, REAL, NULL etc.). Novice, intermediate and expert subjects were asked to learn each of the words presented on the cards such that they could subsequently recall them without the aid of the cards. Subjects were then asked to attempt to recall the words they had learnt during 25 scheduled trails. Some of these trials were cued in that subjects were presented with a reserved word and were then asked to recall other words "that go with it". Most of the trials, however, were uncued and subjects could recall words in the order they wished.

McKiethen et al. analysed the recall order for each subject using an algorithm developed by Reitman and Rueter (1980). This algorithm involves searching subject's recall strings for all groups of items that always appear contiguously, regardless of recall order. For example, a number of groups of contiguously recalled items (ABC, BC, DEFG, DEF, EFG, DE, EF and FG) can be derived from the four recall strings in figure 3.1. These groups can be represented either as a lattice under set inclusion or as a parenthesised expression. Consistency in the order of recall of groups (such as an ordered set of chunks or a list) appears in the lattice as overlapping groups: e.g., DEFG, DEF, EFG, DE, EF and FG indicate that DEFG is always ordered. Inspection of the original recall strings can determine whether the order is always the same (i.e., a unidirectional chunk) or one order and its reverse (a bidirectional chunk). Unidirectional chunks can be represented in the tree diagrams by using a single-headed arrow notation, and bidirectional chunks by double-headed arrows. In the parenthetic representation, square brackets denote a unidirectional chunk and angle brackets a bidirectional chunk. Nondirectional chunks, whose constituents can appear in any order, are indicated by the absence of an arrow in the tree diagram and by curved brackets in the parenthetic expression.

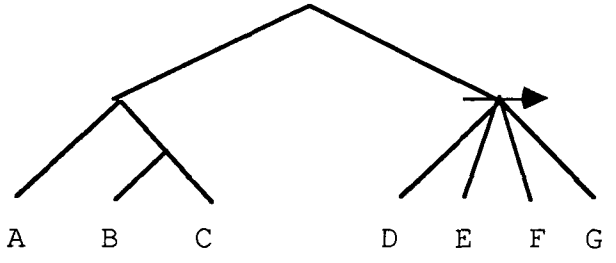
a. Recall Strings

A B C D E F G
D E F G C B A
D E F G B C A
B C A D E F G

b. Lattice of Chunks



c. Ordered Tree



d. Expression

((A (B C))[d e f g])

Figure 3.1. Example analysis of recall strings by the Reitman - Rueter technique.

Figure 3.2 shows the organisation of reserved words for a novice programmer and for a typical expert programmer. McKiethen et al. found that beginning programmers appear to utilise a range of different mnenomic techniques. For instance, some novice subjects appeared to be using an encoding strategy based upon orthography, while others seemed to be using common natural language sequences such as DO-FOR-WHILE or BITS-OF-STRING. Experts, on the other hand, appear to cluster together words based upon common structures used in ALGOL (i.e., WHILE-DO or IF, THEN, ELSE), while intermediates appear to employ a mixture of strategies and to base their organisation on both ALGOL structures and natural language chunks.

The results of the McKeithen et al. study provide an impressive demonstration that differences in knowledge organisation, as reflected by recall order and memorisation strategy, are strongly correlated with differences in programming skill. McKeithen et al. do not claim that particular forms of knowledge organisation produce expertise, since any support for this hypothesis would need to be derived from a demonstration of within-subject skill changes with instruction. However, one might hypothesise that learning particular forms of organisation would led to the enhancement of programming skill.

Moreover, McKeithen et al. suggest that their findings may support the claims made by protagonists of structured programming, since it is possible that an expert's mental organisation of a program may correspond to the forms of organisation produced by applying structured programming techniques. Hence, it appears that the Reitman - Rueter technique for analysing recall order that is adopted by McKeithen et al. may provide the basis for exploring the purported cognitive advantages of adopting structured programming principles. The main contribution of the McKeithen et al. study is that it provides evidence for the way in which programmers group together related pieces of information in order to form knowledge structures that can best guide problem solving in programming. Their work provides a natural extension to some of the earlier chunking studies by outlining the different strategies used to group information and by illustrating the way in which particular strategies are associated with different levels of skill.

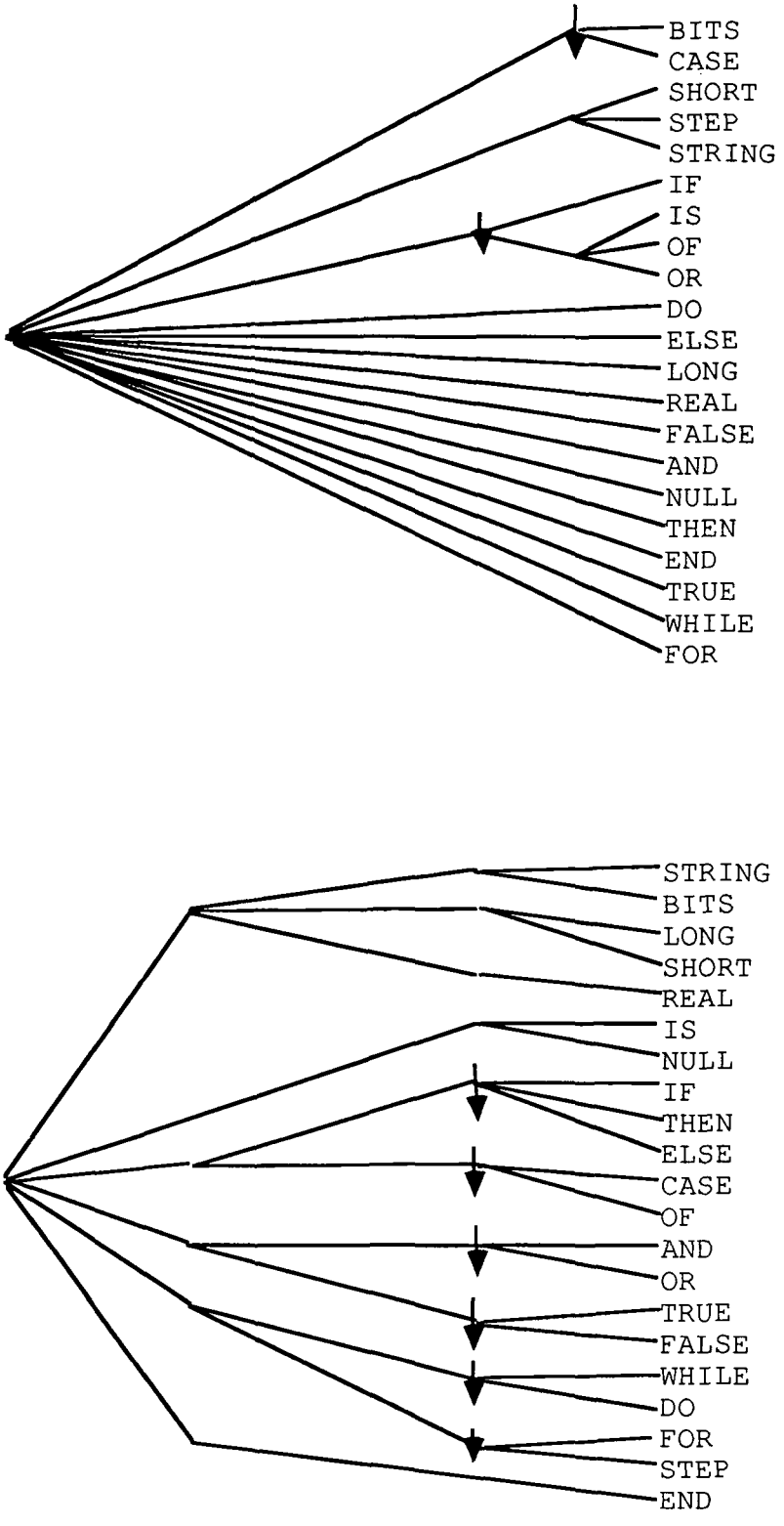


Figure 3.2. Typical novice (above) and expert (below) knowledge organisation found in the McKeithen et al study. The novice's organisation is apparently based on orthography, while the expert's is based upon the meaning of words in the programming language.

The results of the McKeithen et al study suggest that experts group program statements according to broad semantic categories while novices use familiar surface features such orthography. In other problem solving domains similar finding have emerged. For instance in Physics problem solving it has been shown that novices typically represent only the surface (or concrete) features of a problem whereas experts represent abstract physical principles (Chi, Glaser and Rees, 1981; Chi, Feltovich and Glaser, 1981). Similar findings have also emerged in studies of mathematics (Schoenfeld and Herrmann, 1982).

3.3.2 Abstract and concrete representations of programming knowledge

In the programming domain, Adelson (1981) has attempted to extend these findings by characterising some of the properties of the abstract and concrete representations that are formed during program comprehension. Adelson (1981) found that expert programmers used abstract conceptually based representations when attempting to recall programs, whereas novices used syntactically based representations. Using a multi-trial free recall procedure, Adelson asked novice and expert programmers to recall 16 lines of program code that had been presented randomly. Although the subjects had not been told that the 16 lines could be organised either conceptually into three programs or syntactically into five categories according to the control words they contained, an analysis of the recall for each group showed that experts had clustered the lines into complete programs, while the novices clustered lines according to syntactic category. This finding simply serves to replicate the results of the McKiethen et al (1981) study reported above. Adelson (1984; 1985) later extended this basic paradigm to explore in more detail the nature of the organisational groupings produced by expert and novice programmers, by characterising some of the properties of abstract and concrete representations.

For example, Adelson (1984), reports an experiment in which expert and novice subjects were given tasks that required them to form and use both abstract and concrete representations. Subjects were presented with a stimulus set consisting of eight PPL (Polymorphic Programming Language, described as similar to APL and PL/1) programs with two types of flowchart and two types of questions for each of the eight programs. The flowcharts were constructed such that one described the output resulting from program execution, while the other described how the program functioned.

Adelson suggests that flowcharts that represent what a program does (i.e., in terms of its expected output) should be considered as abstract representations, since they describe the results obtained without specifying the method used to achieve them (e.g., in order to sort a set of items alphabetically one might use a variety of methods such as a merge sort, shell sort or bubble sort). Conversely, the flowcharts that describe a program's function are referred to as concrete in the sense that they represent procedural information without providing any general descriptive information about expected results. Adelson suggests that these two categories of abstract and concrete representation have strong parallels to the more common distinction that is drawn between procedural and declarative knowledge (Anderson, 1983; Winograd, 1974).

In addition to being presented with a program and a flowchart describing that program (Figures 3.3 and 3.4), subjects were presented with a number of questions relating to either concrete or abstract features of the program. For example - "Is the field wider than it is long?" (abstract question, what the program does), "which border of the field is filled in first?" (concrete question, how the program works).

The level of abstraction of the flowchart was crossed with the level of abstraction of the question to create four conditions for subjects of different levels of expertise: two appropriate set conditions, in which the level of abstraction of the flowchart matched the level of abstraction of the question, and two inappropriate set conditions where the level of abstraction of the flowchart did not match the level of abstraction of the question. Hence, in the appropriate set conditions, subjects saw either an abstract flowchart followed by an abstract question or a concrete flowchart followed by a concrete question. In the two inappropriate conditions, subjects were presented with either an abstract flowchart followed by a concrete question or a concrete flowchart followed by an abstract question. The two dependent variables used in this experiment were comprehension time (the time it took a subject to state that they had understood the flowchart well enough to go on and study the program and answer the associated question) and error rates on the questions.

```
PROGRAM

$MAKEFIELD; I;J

[1] ... DATA DEFINITIONS IN THE CURRENT ENVIRONMENT
[2] ... $GRID = [1 : ] ROW
[3] ... $ROW = [1 : ] CHAR
[4] LINE - MAKE (ROW, 40,')              ...CREATE THE FIELD
[5] FIELD - MAKE (GRID, 20, LINE)
[6] FOR I - 1:19:20 DO %                  ...FILL IN TOP AND BOTTOM
    (FOR J - 1:40 DO FIELD [I,J] - 'B)    ...BORDER
[7] FOR I - 2:19 DO %
    (FOR J - 1:39:40 DO FIELD [I,J] - 'B  ...FILL IN SIDES
```

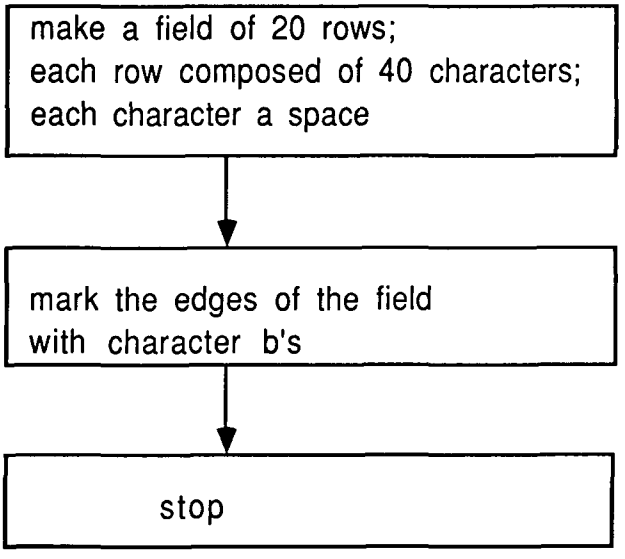


Figure 3.3. Program described in the following flowcharts. The flowchart above is an abstract flowchart in that it describes what the program does.

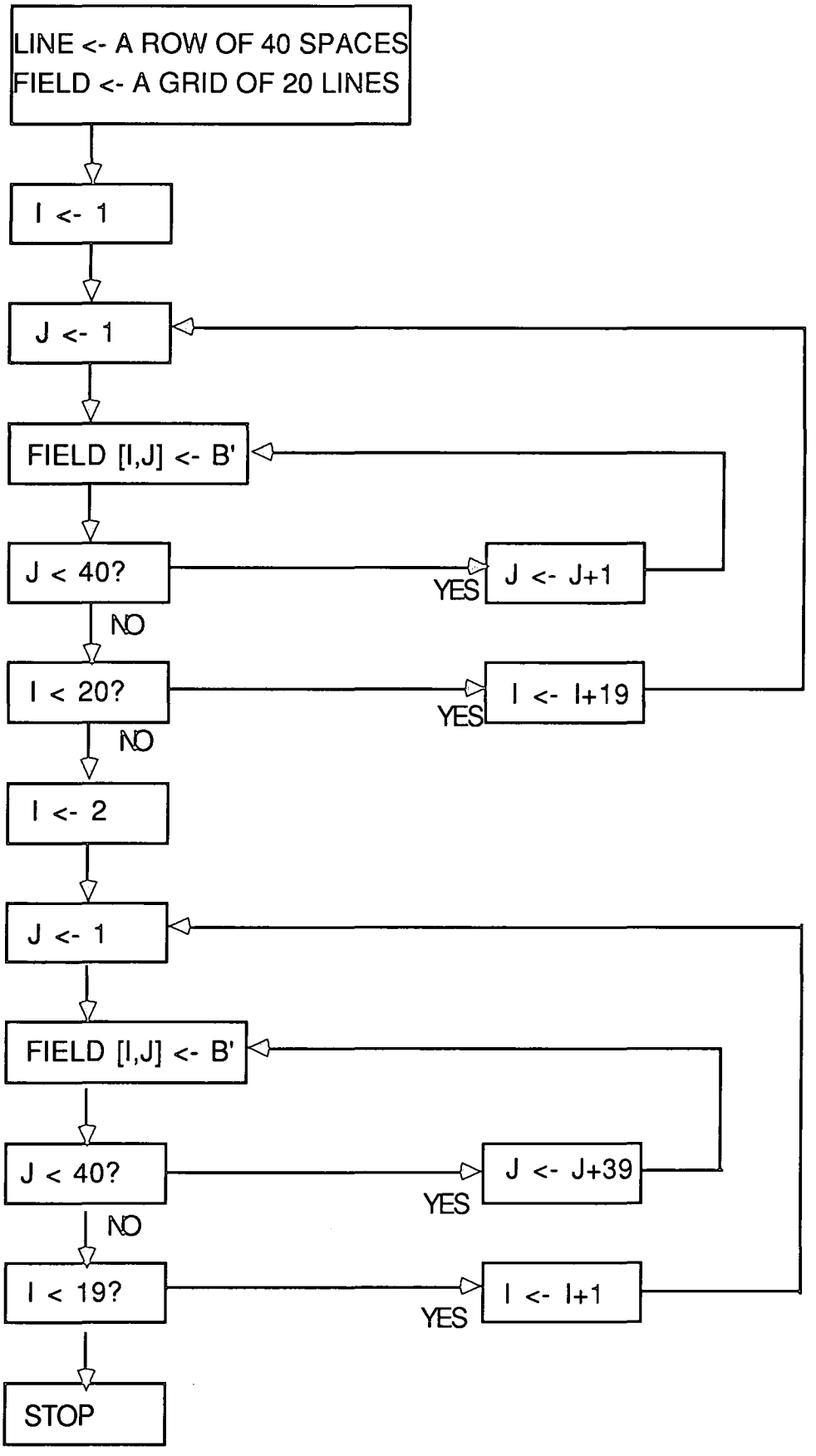


Figure 3.4. An example of a concrete flowchart .

Adelson found that both types of flowchart reduced program comprehension time significantly. In addition, the abstract flowcharts were comprehended more quickly than the concrete flowcharts. No differences in comprehension times were evident between the novice and expert groups for the concrete/abstract conditions. In terms of the data relating to errors, Adelson found that experts performed more accurately in the inappropriate set condition in response to concrete questions while novices performed more accurately in response to the abstract questions. In the appropriate set conditions, experts answered both the concrete and the abstract questions more accurately than novices.

Adelson suggests that these results support the idea that novice programmers represent the concrete or procedural aspects of a problem while experts tend to form more abstract declarative representations. It is worth noting here the contrast between Adelson's findings and Anderson's suggestion that experts employ procedural knowledge and novices, declarative (see chapter 2). In the inappropriate set conditions, where the level of abstraction of the flowchart and the question do not match, the flowcharts will inappropriately prepare subjects to represent the program at one level of abstraction or the other. Adelson reasons that if the level that is actually appropriate is the subjects' natural or preferred level, then they would still be able to perform well. That is they would be able to accurately and quickly form a representation of the program that is appropriate to answering a particular question, and be able to process the information contained in this representation despite the misleading set. Conversely, if the appropriate level is not natural to the subject, the effects of the inappropriate set condition plus the non-naturalness of the representation will combine to impair performance.

Adelson claims that the patterns of results in the appropriate set conditions provide information about the natural level of representation for each group. Hence, a condition in which a group's performance is good is a condition in which the required representation is natural, and a condition in which a group's performance is poor is one in which the required representation is not natural. Hence, since the novice group perform better in the concrete condition, while the expert group perform more accurately in the abstract condition we can infer that novices represent programming knowledge in concrete terms and experts in abstract fashion.

This finding provides additional support for the idea that novices and experts represent domain knowledge differently. In programming, experts appear to focus upon the declarative aspects of a problem while novices place emphasis upon a problems procedural aspects. Some support for this distinction can also be derived from work in other domains, suggesting that this finding may be generalisable to other areas of expertise. For instance, Brown and Burton (1978) found that school mathematics teachers who were expert in performing elementary arithmetic operations, experienced difficulty verbalising these operations.

The importance of different forms of representation is strikingly evident in Adelson's (1984) study in the context of those conditions where novice performance actually surpasses that of experts. Adelson claims that experts have learnt that during comprehension it is more profitable to focus upon high-level abstract elements of a program rather than on low-level implementation details. She claims that this point will also be valid for very high-level languages in which the distinction between what is abstract and what is concrete may not map onto the distinction between what a program does and how it functions. However, Adelson provides no evidence for this conjecture, and more recent research (Cooke and Schvaneveldt, 1988; Gilmore and Green, 1988) has suggested that the cognitive structures that have been observed in this and similar studies may be specific to the particular programming language used in the studies.

3.4 Are representations of programming knowledge language independent?

Cooke and Schvaneveldt (1988) have observed that the stimulus material used by Adelson, and for that matter by the majority of researchers studying programming expertise, has consisted of either program statements or reserved programming words. They suggest that this kind of material does not provide an ideal means of eliciting information about a subject's semantic knowledge which should, by definition, be language independent. This would be predicted by other models of programming. For instance, according to Shneiderman's (1980) syntactic/semantic model, syntactic knowledge is specific to each programming language, while semantic knowledge represents general programming concepts that are not language-specific.

Cooke and Schvaneveldt (1988) also suggest another important methodological limitation of studies of programming expertise. They argue that these studies have

tended to categorise individuals with some programming expertise as the least experienced programmers, thus ignoring completely naive programmers with no programming experience at all. Cooke and Schvaneveldt suggest that the cognitive structures of individuals prior to any exposure to domain-related material may reveal preconceived notions, misconceptions or prior knowledge that might be of interest to those involved or interested in education. Presumably these preconceived notions etc. may play an important role in the development of programming skill and knowledge representation.

Cooke and Schvaneveldt (1988) report a study in which they attempted to address some of the limitations of previous work that have been outlined above. In particular, their study examined the organisation of a set of abstract programming concepts for subjects of various levels of skill, including a naive group who possessed no programming experience whatsoever. Cooke and Schvaneveldt (1988) presented their subjects with a set of 16 programming concepts derived from an introductory computer science text book. Subjects were asked to assign ratings, on a scale of zero to nine, to each concept based upon their familiarity with that concept. The subjects were then asked to undertake a relatedness rating task, where they were informed that there are a number of dimensions along which the concepts could be related (e.g., frequency of co-occurrence, generality, importance). An assignment of a zero score for a pair of concepts indicated that the pair were highly unrelated, while a score of nine indicated that the pair were highly related.

These data were then used to construct network representations that indicated the strength of the relatedness measure coexisting between concepts. These networks were constructed using the Pathfinder algorithm (Dearholt, Schvaneveldt and Durso, 1985; Schvaneveldt, Durso and Dearholt, 1985) which produces a network, from empirically derived relatedness measures, in which items or concepts are represented as nodes and the relationships between items are represented as links between nodes.

A weight corresponding to the strength of the relationship between nodes is associated with each link and is taken to represent the inverse 'psychological' distance between nodes. The Pathfinder scaling technique has been applied to a number of domains including the design of menu systems (Roske-Hofstrand and Paap, 1986). A number of studies have shown that Pathfinder networks are psychologically meaningful. For instance, such networks have been shown to

have predictive value in that they can account for a subjects free recall performance better than other scaling techniques (Cooke, Durso and Schvaneveldt, 1986).

The results of the Cooke and Schvaneveldt study indicated that the networks that were derived from the estimates of relatedness performed by their four experimental groups differed systematically with experience. The naive group appeared to base their structuring upon the meaning of the terms in natural language. This provides additional support for the finding of McKiethen et al who demonstrated that natural language associations are instrumental in the organisation of recall for novice programmers. In addition, the intragroup correlations between networks produced from the naive subjects' relatedness estimates proved to be high, suggesting a shared conceptual structure. In the case of the advanced group, conceptual organisation appeared to be based upon the meaning of the concepts in the programming domain. The intermediate network representation was rather similar to the network obtained for advanced subjects, in that the two networks shared many common features. One of the more interesting findings of this study was that the relatedness rating correlations demonstrated that intragroup agreement did not increase in a linear fashion with increasing expertise. Rather, agreement tended to decrease from the naive to the novice level and then gradually increase from the novice to the advanced level.

The fact that naive subjects shared some common domain knowledge, albeit inappropriate to the programming task itself, is interesting in that it suggests that naive subjects have a mental model of programming just as the non-expert may have a naive mental model of physics (Gentner and Gentner, 1983). In other non-programming domains it has been demonstrated that inappropriate models can lead to serious conceptual difficulties during learning (Norman, 1983). The results of the Cooke and Schvaneveldt study suggest a means of identifying the source of beginning programmers misconceptions and thereby may contribute to instructional practice by explicitly focusing on the formation of appropriate conceptualisations. It is interesting to note that subjects with some, albeit limited, programming experience (i.e., the novice and intermediate groups) did not share a common conceptual structure.

Cooke and Schvaneveldt suggest that this decline in agreement may be due to variations in teaching strategy, the use of particular text books etc. However, once concepts become well-learned, these variations in programming experience no longer appear to matter. They suggest that the advanced network representation

can be considered to be an explicit goal state in the learning process, while the naive representation corresponds to the state prior to learning. An analysis of intermediate states may indicate conceptual misunderstandings that have arisen during the learning process and Cooke and Schvaneveldt suggest that this will have implications for the adoption of particular teaching or training strategies.

The studies that have been reviewed above have a number of implications for our understanding of the way in which knowledge representation changes with increasing expertise. In particular, they indicate that expert and novice representations have some very specific but differingproperties. For instance, experts have been shown to group information in a semantic fashion, while novices group information according to the surface features exhibited by a program. Adelson has demonstrated that experts tend to represent the declarative information relevant to a particular program, while novices focus to a greater extent upon a program's procedural content.

While these studies have been important in terms of describing the more general features of programmers' knowledge representations, they do not provide an indication of the detailed content of these knowledge structures. There is, however, a growing body of literature that has focused explicitly upon a description of the kinds of tacit knowledge that programmers appear to be able to recruit to guide problem-solving. These studies have drawn analogies with work in text comprehension to suggest that problem-solving in programming is mediated by the possession of script or schema-like knowledge structures that provide specific techniques for commonly occurring programming procedures that are both language and problem independent.

3.5 Knowledge Representation in programming: Analogies with text comprehension

3.5.1 Introduction

A number of important analogies have been drawn between work in the text comprehension domain and theories of knowledge representation in programming. Clearly there are certain common features of these domains that make this analogy plausible. For instance, programming involves the construction of text-like structures which are used to instruct a computer. The use of programming

languages might therefore been seen to be similar to our use of the written word for communicative purposes[1].

Here, perhaps the most important analogy is with notion of a 'schema'. Schemas are proposed as knowledge structures which consist of a set of propositions that are organised in terms of their semantic content. There are two primary principles upon which most schema theories are built. Firstly, that cognitive processing is guided and limited by the application of prior knowledge. Secondly, that schemas contain relatively abstract knowledge which is largely independent of any one event. Moreover, in most theoretical accounts of schema use such structures are thought to be organised hierarchically, and to facilitate reasoning via the instantiation of default values in situations where information is not present in the task domain.

Schema-based mechanisms for cognitive control offer a method for limiting the amount of inputted information, or bottom-up control, that is needed to perform a task. Moreover, schemas provide top-down control by using prior knowledge to restrict the range of possible operations that might be undertaken. Hence, either a perceptual input or a cognitive goal or process may evoke a schema with a related semantic content.

Such theories have appeared in a number of forms. For instance Minsky, from an AI perspective, uses the term 'frame' (Minsky, 1971). Schank and Abelson identify a particular type of schema known as a 'script' composed of a sequence of abstracted actions which occur in the context of common events, with slots for specific instances (Schank and Abelson, 1977). Schank and Abelson also identify 'plans', which are executed in order to determine the inferences that are required in order to understand situations for which there are no stereotypical event sequences. Hence, scrips denote sequences of actions that have occured on numerous occasions (e.g. visting a restaurant), whereas plans describe the production of novel action sequences (e.g. robbing a bank).

Schema theories have been used to account for a wide range of cognitive behavours, such as language understanding, memory and problem solving. As such, we might regard such theories as fairly powerful, however there is a danger that this explanatory potential may result in the concept becoming too nebulous to be of any real value as a predictive mechanism. Too some extent this problem is reflected in some of the criticisms of schema-based theories of programming that

are proposed by this thesis. In particular, it is argued that schema theories of programming knowledge must specify explicity the mechanisms that mediate the acquisition and use of programming knowledge.

Soloway and his colleagues (Detienne and Soloway, 1990; Soloway and Ehrlich, 1984; Soloway, Ehrlich, Bonar and Greenspan, 1982; Soloway, Ehrlich and Gold, 1983; Spohrer, Soloway and Pope, 1985), drawing upon the work of Schank and others (Schank, 1980; 1981; Schank and Abelson, 1977), have proposed that programmers possess and are able to access abstract schematic plan-based structures that represent stereotypical programming knowledge. A number of other authors have attempted to provide a more formal specification of plan knowledge in programming. For instance Rich, Shrobe and Waters (Rich, 1981; Rich, Shrobe and Waters, 1979; Waters, 1979; 1982) have developed a large collection of plans based upon their intuitions about programming. It is likely that there are hundreds (possibly thousands) of these plans which can be used to guide problem-solving in programming, and as in other domains this repertoire of plans provides a set of standard methods for achieving certain types of goals.

Ehrlich and Soloway (1984) suggest that such plans provide high-level structures that serve to chunk together related pieces of information. Such plan structures are regarded as similar to Schank and Abelson's (1977) notion of scripts (See above). Scripts are used to explain how people can understand stereotypic sequences of events such as eating at a restaurant or going to a doctor. By analogy to this, Ehrlich and Soloway claim that plan knowledge in programming consists of a catalogue of stereotypic action sequences. These action sequences describe the programmer's tacit knowledge of the domain. The possession of such tacit knowledge structures has been shown to be an important factor in distinguishing experts from novices in other domains (Collins, 1978, Larkin et al, 1980; Polya, 1973) and it would seem reasonable to suggest that expert programmers are able to recruit similar knowledge in order to guide their problem solving activities. Previous studies of the programming activity, such as those described above, have sought to establish that experts not only have more knowledge about programming than their less experienced counterparts, but that experts can be distinguished from novices in terms of their better organisation of knowledge. The work of Soloway and his colleagues has extended previous work by describing the typical content of expert knowledge structures and as such has provided a valuable insight into the way in which tacit knowledge can guide problem solving activities.

3.5.2 Programming Plans and Discourse Rules

Soloway and Ehrlich (1984) introduce the notion of the *programming plan* to provide a description of those program fragments that are thought to represent stereotypic action sequences in programming. They suggest that expert programmers possess two main kinds of plan: Plans that relate to aspects of a program's control flow and plans that represent facets of variable use. An example of a control flow plan might be a plan that accumulates and keeps track of a running total (see figure 3.5). Soloway and Ehrlich suggest that such a plan might be used in variety of programs which may have been constructed to implement solutions to a wide range of problems. In this sense, such plans should be regarded as both language and problem independent. Hence, programs are constructed on the basis of generalised plan knowledge and specific plans are created in response to the requirements and constraints of a particular problem. The use of the term plan may seem rather odd here, since as we have seen, plans are usually taken to denote the action sequences that are required in the context of novel events. As we remarked above, perhaps a more useful description would equate the notion of plans with the concept of a 'script'.

The composition of plans, according to Soloway and Ehrlich, is mediated by so-called rules of programming discourse (see also, Leventhal, 1987; 1988), which are proposed to be directly analogous to conversational discourse rules. An example of a program discourse rule is that variable names should normally reflect their function[2]. Hence, a program might be correct in that it produces the right results but may be difficult to read because it violates certain rules of discourse.

3.5.3 Empirical Studies

Soloway and Ehrlich (1984) report a number of empirical studies to support their contention that programmers are able to recruit tacit plan knowledge during problem solving and that the possession of such plan knowledge can be used to distinguish experts from novices. These studies employ two major experimental paradigms. Firstly, a fill-in-the-blank task, where subjects are presented with a program fragment with one or more lines omitted, their task being to supply the missing line/s and secondly a straightforward free recall task. Both sets of studies compared performance in these two tasks in response to the presentation of plan or unplan-like programs.

In this context, a plan-like program was constructed such that its component plan structures were consistent with certain rules of programming discourse. An unplan-like version (Soloway and Ehrlich's term) of the same program can be constructed by introducing violations to one or more of its consituent plan structures (see figure 3.6). It should be noted that in all cases both the plan and unplan-like versions were executable programs, and in the majority of cases computed the same values. This can be contrasted with the experimental materials used in a number of earlier studies which consisted of program statements presented in random order.

Version A

```
PROGRAM Yellow (input, output),
VAR I  INTEGER
   Letter, LeastLetter    Char,
BEGIN
   LeastLetter = 'a',
   FOR I = 1 TO 10 DO
      BEGIN
         READLN (Letter),
         If Letter > LeastLetter
            THEN Leastletter = Letter,
      END,
   Writeln (Leastletter),
END
```

Version B

```
PROGRAM Green (input, output),
VAR I  INTEGER
   Letter, LeastLetter    Char,
BEGIN
   LeastLetter = 'z',
   FOR I = 1 TO 10 DO
      BEGIN
         READLN (Letter),
         If Letter < LeastLetter
            THEN Leastletter = Letter,
      END,
   Writeln (Leastletter),
END
```

Figure 3.5 a. These programs both represent a search plan, however in the second case (Version b) the program violates the discourse rule which suggests that "a variable name should reflect its function".

Version A

```
PROGRAM Pink (Input, Output),
   CONST
      MaxSentence =  99,
      NumOfConvicts = 5,
   VAR
      ConvictID, I, Sentence    INTEGER
   BEGIN
      FOR I = 1 to NumOfConvicts DO
         BEGIN
            READLN (ConvictID, Sentence),
            IF Sentence > MaxSentence
               THEN Sentence = MaxSentence,
            WRITEIN (ConvictID, Sentence),
         END
END
```

Version B

```
PROGRAM Gold (Input, Output),
   CONST
      MaxSentence =  99,
      NumOfConvicts = 5,
   VAR
      ConvictID, I, Sentence    INTEGER
   BEGIN
      FOR I = 1 to NumOfConvicts DO
         BEGIN
            READLN (ConvictID, Sentence),
            WHILE Sentence > MaxSentence
               DO Sentence = MaxSentence,
            WRITEIN (ConvictID, Sentence),
         END
END
```

Figure 3.5 b. The basic plan represented by this program involves resetting variables to boundary conditions. The discourse rule violated in version B is "An IF should be used when a statement body is guaranteed to be executed only once and a WHILE used when a statement body may need to be executed repeatedly".

Fill-in-the-blank tasks

In their first study, Soloway and Ehrlich presented novice and advanced programmers with a number of plan-like and unplan-like program fragments in which certain critical elements had been omitted (figure 3.6). The subject's task was to fill in the blank line with a piece of code that they thought best completed the program. According to Soloway and Ehrlich, subjects who engage in this task will need to infer the intention of the program based upon their knowledge of the problem and upon their tacit plan knowledge and this should create expectations about what would constitute an appropriate way of completing the program. A similar technique has been used in text comprehension work in order to explore subjects' underlying knowledge of typical real world events (Bower, Black and Turner, 1979; Kemper, 1982) and these studies suggest that this technique does provide an appropriate means of eliciting subjects' tacit knowledge about a particular domain.

Soloway and Ehrlich reason that if advanced programmers possess and use programming plans then they should be able to recognise program fragments in plan-like versions of programs as examples of particular plans and consequently they will complete the blank line in that program with code that corresponds to the role expressed by that plan. In the case of unplan-like programs, the advanced programmer will not be able to infer the program's plan structure and they will consequently be less likely to fill in the missing line correctly. Novice programmers, who, it is claimed, have not developed a full repertoire of plans and programming conventions, will not be guided by plan structures and hence should perform with similar levels of accuracy in response to both plan-like and unplan-like programs.

Soloway and Ehrlich's results provide support for these hypotheses. Firstly, experts were able to complete blank lines in programs more accurately than novices (61% correct response vs 48% correct response). In addition, subjects completed the plan-like versions correctly more often than the non-plan like versions. Finally, there was a significant interaction between program version (plan-like or unplan-like) and expertise.

Version A

```
PROGRAM Brown (input, output),
 VAR Num    REAL,
    I    INTEGER,
 BEGIN
    FOR I = 1 TO 10 DO
       BEGIN
          Read (Num),
          IF Num < 0 THEN Num = -Num,
          Writeln (Num, Sqrt (Num)),
       END
 END
```

```
PROGRAM Brown (input, output),
 VAR Num    REAL,
    I    INTEGER,
 BEGIN
    FOR I = 1 TO 10 DO
       BEGIN
          **********
          IF Num < 0 THEN Num =-Num,
         Writeln (Num, Sqrt (Num)),
       END
 END
```

Version B

```
PROGRAM Green (input, output),
 VAR Num    REAL,
    I    INTEGER,
 BEGIN
    Num = 0
    FOR I = 1 TO 10 DO
       BEGIN
          Read (Num),
          IF Num < 0 THEN Num = -Num,
          Writeln (Num, Sqrt (Num)),
       END
 END
```

```
PROGRAM Green (input, output),
 VAR Num    REAL,
    I    INTEGER,
 BEGIN
    Num = 0
    FOR I = 1 TO 10 DO
       BEGIN
          **********
          IF Num < 0 THEN Num =-Num,
         Writeln (Num, Sqrt (Num)),
       END
 END
```

Figure 3.6. An example of the experimental materials used by Soloway and Ehrlich (1984). The basic plans in this example are a guard plan and a variable plan. Version a is plan-like, while version b is unplan-like in that it includes two incompatible discourse rules.

Moreover, the difference in performance between the novice and expert subjects for plan-like programs was significant, while in the case of unplan-like programs, no difference was evident. Indeed, in the case of unplan-like programs, the performance of the expert group was reduced to that exhibited by the novice group. These results provide strong support for the idea that expert programmers use plan knowledge to guide their comprehension of programs. This knowledge provides a basis upon which expectations about program function can be constructed. Moreover, when these expectations are violated, expert performance is reduced drastically.

Program Recall

In a second study, Soloway and Ehrlich examined subjects' recall of plan-like and unplan-like programs. Soloway and Ehrlich employed the same stimulus materials in this study as in their fill-in-the-blank tasks, however only expert programmers participated. Subjects were presented with a program which they were subsequently asked to recall verbatim. Half of these programs were plan-like and the other half were unplan-like. Figure 3.6 shows examples of the programs used in this study. Notice that the programs are identical except for two 'critical' lines. These lines are described as critical in the sense that they convey information as to whether the program is plan-like or unplan-like. Subjects were presented with a program three times. During the first trial subjects were requested to recall as much of the program as possible. During the second and third trials, they were asked to either add to their original recall or change any part of their recall that they felt was in error.

They key prediction made by Soloway and Ehrlich is that programmers would recall the critical lines from plan-like programs earlier than the critical lines from the unplan-like programs. The idea that the representatives of a particular category are recalled first in free recall studies is a well documented psychological principle (Crowder, 1976). Soloway and Ehrlich, employing this principle, suggest that if expert programmers make use of tacit plan knowledge and discourse rules to encode a program when it is presented, then the critical lines in plan-like programs will be recalled early in a free recall task, since these lines are considered to be the key representatives of a particular plan. In the case of unplan-like programs, critical lines do not bear a relationship to the program's plan structure and hence are of the same level of significance as other lines in the program. Hence, one would not expect these lines to be recalled any earlier than others.

Once again Soloway and Ehrlich's results provide support for the idea that the problem solving activities of expert programmers are mediated by the possession of stereotypical plan structures. The results from their recall experiment showed that subjects recalled more critical lines from plan-like programs than from unplan-like programs. There was also a significant interaction between version and trial, suggesting that the critical lines in plan-like programs are recalled earlier than in the unplan-like programs.

In summary, the idea that plan knowledge plays an organising role in memory suggests a number of features that appear to be central to program comprehension. Firstly, the comprehension process appears to proceed by the recognition of patterns that implement known plans. Secondly, plans will be activated by partial pattern matches and confirming details will either be sought or assumed. Hence, plan structures are seen to guide problem-solving via both the application of known methods and through the creation of expectations about the typical form and behaviour of these methods.

3.5.4 Other work on plans

Interactions between everyday knowledge and programming language constructs

The close analogy between studies of schema theory in text comprehension and in programming research has led to a number of other predictions about the role of tacit knowledge in programming. For instance, work in the text comprehension domain carried out by Bower et al. (1979) has shown that when texts violate the stereotypical sequences dictated by a script structure, subjects will tend to reorder the text during recall tasks so that it conforms to structures suggested by their tacit real world knowledge. In addition, Schank (1979) has proposed that the inferences that are typically made during text comprehension are affected by schema-congruent expectations. According to Schank, the salience of a particular text structure to a given reader is partly determined by its relative congruity or incongruity with that reader's schematic knowledge.

In a similar vein, Soloway, Bonar and Ehrlich (1983) found that programmers show a tendency to order program statements in a manner dictated by their everyday knowledge even though this ordering leads to bugs in their programs.

For instance, this study showed that the process/read loop construct in Pascal (see figure 3.7) is a major source of bugs because it mismatches the normal course of events in the real world, i.e., normally we would first need to get an object (read it) in order to process that object in some way.

```
program E1;
   var Count, Sum, Number : integer;
      Average ; real;
  begin
      Count := 0;
      Sum := 0;
      Read (Number);
      while Number <> 99999 do
         begin
            Sum := Sum + Number;
            Count : = Count + 1;
            Read (Number)
         end;
      if count > 0
         then
           begin
                Average := Sum/Count;
              Writeln (Average);
           end
       else

           Writeln (' no numbers input:
                    average undefined');
   end
```

```
program Pascal L;
   var Count, Sum, NewValue: integer;
      Average: Real;
   begin
      Count := 0;
      Sum := 0;
      loop
         Read (newValue);
         if NewValue = 99999 then leave:
         Sum := Sum + NewValue;
         Count := Count + 1;
      again
      if Count  > 0
         then
            begin
               Average := Sum/Count
                     Writeln (Average);
          end
      else
               Writeln ('no numbers input:
                        average undefined')
end
```

Figure 3.7 a. Two programs illustrating different looping strategies. Program E1 is a normal Pascal program, utilising a PROCESS/READ strategy, while the program on the right is written in Pascal L and uses a READ/PROCESS strategy.

```
Read (first value)
while Test (ith value)
   do begin
      Process (ith value)
      Read (i + 1st value)
   end
```

```
loop
   do begin
      Read (ith value)
      Test (ith value)
      Process (ith value)
   end
```

Figure 3.7 b. Schematic representations of, on the left, the process/read strategy typical of Pascal, and on the right, the read/process strategy embodied in Pascal L.

In addition, this study showed that subjects wrote correct programs more often when they used a language that facilitated their preferred (i.e., read then process) cognitive strategy. For instance, the language Pascal L uses a loop...leave..again construct (see figure 3.7) and this language appears to facilitate the construction correct programs for intermediate and expert programmers. Additional support for the idea that everyday knowledge can create misleading expectations for beginning programmers is provided by Eisenstadt, Breuker and Evertsz (1984) who demonstrated that novice programmers can recruit plans based upon 'real-world' algorithms, but that difficulties arise when there is not a straightforward mapping between these algorithms and the kinds of operations that are allowed in a particular programming language.

Plan violation and program complexity

Additional support for the role of plan structures in comprehension has been derived from the idea that if plan structures are a key feature in program comprehension - the thesis suggested by Soloway and Ehrlich, and supported by their empirical research - then the extent to which a program violates normal plan composition should provide a measure of the understandability of that program. In order to explore this hypothesis further, Soloway, Ehrlich and Black (1983) compared three program analysis techniques - Halstead's Volume metric (Halstead, 1977), propositional analysis (Atwood and Ramsey, 1978) and plan analysis (Soloway and Ehrlich, 1984) - in order to determine the extent to which a measure of plan violation can predict program comprehension.

Soloway et al (1983) took three versions of a program intended to solve the same problem and subjected the programs to the three analysis techniques mentioned above. The main difference between the three versions of the program was that the plan composition in two of the programs violated a number of rules of programming discourse. Soloway et al suggest that programmers will expect other programmers to follow these rules of discourse. Hence, plan or discourse rule violations will make programs to more difficult to comprehend. The question that Soloway et al address is whether the results of the three different methods of program analysis they consider can distinguish comprehensible programs (i.e., programs that conform to plan structure) from less comprehensible programs (i.e., programs that violate plan/discourse rule structure).

The first analysis technique, Halstead's Volume Metric, involves calculating the total number of operations (N_1) and operands (N_2) in a program and the number of unique operations (n_1) and operands (n_2). These measures are then combined according to the following equation:

$$V = (N_1 + N_2) \log_2 (n_1 + n_2)$$

The resulting number is intended to provide a measure of the size and broad complexity of a program. In addition, one might assume these factors to be a reasonable predictor of comprehensibility. While some studies have shown that Halstead's Volume Metric can predict programmer performance (eg., Sheppard, Borst and Love, 1978), Soloway et al found that the volume metric calculated for their three programs was equivelent. Hence this metric does not appear to provide a good basis for measuring the relative comprehensibility of different programs.

Next, Soloway et al subjected the three programs to a propositional analysis. This form of analysis was derived from text comprehension work carried out by Kintsch and van Dijk (1978) who have suggested that texts can be decomposed and analysed in terms of their constituent propositional structure. The assumption here is that a text with a complex propositional structure will be more difficult to understand than a text with a simpler propostional structure. In the text comprehension domain there is considerable empirical support for the validity of this assumption (Kintsch and Keenan, 1973; Kintsch, Kozminski, Streby, McKoon and Keenan, 1975). In the context of programming, Soloway et al. suggest that programs will be more complex, and therefore more difficult to understand, when they have a) more propositions (defined by Soloway et al as the composition of a predicate (or operator in the context of programming) and its associated arguments (or operands)) and b) more levels of nesting. However, as with the volume analysis, this propositional analysis failed to distinguish between the plan-like and the unplan-like programs.

This study suggests that plan analysis may constitute a useful method for assessing the comprehesnisbility of programs. The study also provides additional implicit support for the idea that plans play a focal role in program comprehension. Soloway et al readly admit that it is not clear how one might calculate a number that accurately reflects the violation or non-violation of discourse rules[3]. However, they suggest that the plan analysis should be seen as providing

qualitative rather than quantatative information about program comprehensibility, and that a numerical rating system does not typically point out the specific source of program complexity. They suggest that a qualiative analysis can be used to pinpoint specific areas of program complexity and also give a definitive prescription as to how this complexity might be ameliorated.

3.5.6 Plan theory: Automated plan analysis and the development of intelligent tutoring systems

Further evidence for the plan-based view of programming is provided by a study in which novices' errors were analysed in terms of the goal and plan structures inherent in their programs. Johnson, Soloway, Cutler and Draper (1983) characterised bugs by examining the differences between the actual and the intended plan structure of a program. From this they were able to provide a two dimensional analysis of program bugs according to the component of the plan which is affected by the bug and the type of bug or error. For example plan components include, updates, declarations, initialisations etc, while error types include, misplaced, spurious, missing etc. Using this classification technique, Johnson et al were able to analyse successfully 783 novice bugs into 29 of the possible 32 bug categories. This suggests once again that the analysis of programs into plan structures represents a valid framework for exploring the the nature of cognition in programming, since it enables one to predict the kinds a plan-based errors novices are likely to make.

Knowledge-based program analysis - PROUST

This analysis led to the development of PROUST (Johnson and Soloway 1985, Johnson, 1988; 1990; Littman and Soloway, 1988), a system that analyses novices' programs automatically to derive a non-algorithmic, or plan-based, description of the program. PROUST employs a knowledge base of programming plans and strategies, together with a description of the bugs that are commonly associated with these plans and strategies. PROUST carries out an 'intention' based analysis of programs, i.e., it frames its analysis in terms of both the intended functions of the program and in terms of the programmers intention as to how a particular function should to be achieved. According to Johnson, it is difficult to determine the intended function of a program simply by inspecting the

code, since there is no way of knowing whether what a programmer produces is what that programmer had in mind. PROUST derives information about the program's intended function by forming a description of the problem that was assigned to the programmer. In this context, problem descriptions consist of a set of goals to be satisfied and sets of the data objects that these goals apply to.

For instance, for a program which calculates and reports the average daily rainfall over a certain period, the problem description defines a data object, ?DailyRain, which acts as a parameter to a number of the goals in the problem description.One of these goals Sentinel-Controled Input Sequence, forms a goal which requires a series of values to be read, and stops when a specific value (the sentinel) is reached. Figure 3.8 shows the goal decomposition of a problem in PROUST's problem description notation.

From this problem description, the task of identifying the intentions underlying a program involves discovering firstly, how the goals in the problem description relate to the goals that are actually implemented in the program and secondly, what the programmer intended by implementing these goals. PROUST starts by assuming that the student's goals match, or are some variant of, the problem description's goals. If no plausible attempt to implement a particular goal can be found in the code, then PROUST retracts its initial assumption and asserts that the student omitted the goal.

Clearly, programmers might implement the same goal in many different ways. For small programs it may be possible to enumerate all the different ways of implementing a goal, but for more complex problems the variety of different goal implementations will be too great. In PROUST's case the system constructs a description of the intentions underlying each individual student solution. To do this PROUST employs a knowledge base of programming plans. PROUST combines these plans into possible implementations for each goal, and then matches the plans against the code. If PROUST is unable to match its goal decomposition of the problem with its decomposition of the students solution, it attempts to account for this mismatch between the plans and the code. To do this PROUST uses its knowledge base of plan-difference rules, which suggest bugs and misconceptions that may account for the mismatch. For example such a rule might take the following form:

"IF a while statement is found in place of an if statement,
AND the while statement appears inside of another loop
THEN the bug is a while-for-if bug, probably caused by a confusion about the control flow of embedded loops."

Johnson (1988) reports an empirical evaluation of PROUST which gives some idea of its strengths and weaknesses as a system for analysing bugs and misconceptions in novice programs. Figure 3.9 shows the results of running PROUST on a corpus of 206 solution to the same programming problem.

?DailyRain isa Scalar Measurement.

Achieve the following goals:

Sentinel - Controlled Input Sequence (?DailyRain 99999);
Input Validation (?DailyRain, ?DailyRain < 0);
Output (Average (?DailyRain));
Output (Count (?DailyRain));
Output (Guarded Count (?DailyRain, ?DailyRain > 0));
Output (Maximum (?DailyRain));

Figure 3.8. A problem to calculate the average and maximum rainfall over a given time represented in PROUST'S problem description notation.

Total number of programs	206	
Number of programs receiving full analysis:	167	(81%)
Total number of bugs:	598	(75%)
Bugs recognised correctly:	562	(94%)
Bugs not recognised:	36	(6%)
False alarms:	66	
Number of programs receiving partial analyses:	31	(15%)
Total number of bugs:	167	(21%)
Bugs recognised correctly:	61	(37%)
Bugs not reported:	36	(6%)
False alarms:	20	
Number of programs PROUST did not analyse:	9	(4%)
Total number of bugs:	32	(4%)

Figure 3.9. Results of PROUST's analysis of 206 programs written to solve the same problem.

PROUST managed to analyse 81% of the programs completely, that is, it was able to derive a consistent model of the intentions underlying the programs. In these cases, PROUST successfully located 94% of the bugs that were identified by the experimenters. This result is impressive since it demonstrates that PROUST can detect bugs more successfully than traditional manual code inspections such as walkthroughs (Myers, 1978).

Bridge - A plan-based programming tutor

The success of the PROUST system provides additional support for plan-based accounts of program understanding since it demonstrates that plan-based descriptions of programs can provide a good basis for predicting and locating bugs and misconceptions. One natural extension of the PROUST work has involved using its output as a base for an interactive tutorial environment.

Since PROUST runs only in batch mode on a finished program, it is unable to provide a great deal of feedback to students about their progress. However, Bridge (Bonar and Cunningham, 1988; Bonar and Liffick, 1990), a prototype tutorial environment for novice programmers, is able to provide interactive feedback to students of programming. The architecture of the Bridge system will not be discussed in any detail here. However, the philosophy underlying the Bridge approach to plan teaching is germane to our discussion on knowledge-based theories of programming expertise.

The fundamental assumption underlying the Bridge system is that teaching plans to students will improve their basic ability to understand and generate programs. Bonar and Cunningham contrast this with the kind of instruction students receive from programming texts, which introduce a programming language by discussing the syntax and semantics of each statement type . They suggest that this approach exacerbates a common novice tendency to adopt a syntactic matching strategy while problem-solving, such as that observed in physics novices (Chi et al. 1981). For instance, Bonar (1985) showed that novices work linearly through a program, choosing each statement based upon the syntactic features of previously encountered statements. Bonar and Cunningham suggest that one way of overcoming such a syntactic strategy is to teach novices the standard techniques for implementing common tasks, i.e., programming plans.

Bonar and Cunningham have carried out a limited evaluation of Bridge. This evaluation addressed student attitudes to the Bridge system and considered the problems they had interacting with the tutor. However, the evaluation did not directly examine the efficacy of Bridge as a programming tutor. It is interesting to note that while Bonar and Cunningham do suggest that their students were reasonably successful at developing outline solutions using plan-based concepts, they often had some difficulty translating this into a program. They suggest that "Matching between the Phase 2 output and Pascal code was problematic, however. Because there is not always a simple match between a plan component and Pascal code, students will sometimes make a reasonable selection that Bridge doesn't accept". (pg 409). This appears to suggest that knowing plans alone may be inadequate to explain the performance of programmers. Rather, what appears to be important is understanding how plans are used. This brings us to the role strategy in programming expertise and this issue is receives further consideration in chapter 4.

3.5.7 Elaborations of the plan theory - Rist's theory of schema creation in programming

Recently, Rist (1986 a and b; 1989) has attempted to outline the mechanisms that might underpin plan use. Rist has proposed a model of program generation which traces the evolution of a program through a number of stages. An explicit feature of Rist's model concerns the identification of levels of abstraction in program structure. It is claimed that programs are built from simple knowledge structures that are merged and combined to form more complex structures.

At the lowest level of detail , individual fragments of knowledge are combined to form a single line of code. The next stage in the development of a program is to create a programming plan which, as we have seen, provides 'canned' solutions for common goals such as calculating a running total or reading some data value. Next, these plans need to merged into the final program structure. Rist is primarily interested in the processes that underlie the plan generation activity and central to his theoretical explanation is the idea of focal expansion.

Focal expansion describes the process of generating a programming plan from a so-called 'focal line'. In Rist's account, each programming plan has an associated focal line that directly encodes the goal of that plan. For instance a 'running total loop plan' will be associated with the focal line 'count:=count+1'. The complete plan will also consist of an initialisation component and some means of reading data values into the plan. The design of a program is seen to progress through various stages beginning with the implementation of a focal line, its extension to form a complete plan and finally to the creation of an entire program through a process of plan merger.

Rist's model is described in more detail in chapter 4, where an attempt is made to explore the role of strategy in program comprehension and generation. However, there are specific aspects of Rist's model that are germane to the present discussion and in particular his description of the different levels of abstraction in plan-based knowledge representation.

Rist suggests that programming plans can be described as a collection of schemata which are represented as slot-and-filler mechanisms. In addition these schemata represent programming knowledge at various levels of abstraction. For instance,

Programming Plans - or PPlans, in Rist's terminology - represent the lowest level of plan abstraction and are similar to the basic plans described by Soloway and Ehrlich (1984). PPlans have slots for the goal of the plan and for the code that it generates. Complex Program Plans -CCPlans - are built from a number of PPlans to achieve higher-level goals. Rist also describes other types of plans - for instance, Abstract Plans (APlans) which are thought to represent knowledge of different types of loop or sort techniques. Specific Plans (SPlans) represent specific routines such as bubble or shell sort, and Global Plans (GPlans) represent global procedures including "validate", "initialise" or "update".

Rist's work provides a number of important extensions to plan-based theories. Firstly, it suggests a role for different categories and levels of plan knowledge. Whereas the Soloway and Ehrlich description of programming knowledge appears to suggest that plan representations are uniform and internally undifferentiated, Rist's approach asserts that plans can occur at different levels and that certain salient elements of each plan schemata guide the program generation activity. Secondly, Rist makes an explicit attempt to demonstrate how plans are implemented during code generation. One of the major limitations of existing plan theories is that the fail to specify the mechanisms that guide the implementation of plans during coding. In chapter 4, where Rist's work is elaborated in greater detail, the role of schema creation and focal expansion in plan implementation are discussed. These provide the basic mechanisms that control the transformation of plans into code and appear to provide a promising basis for an extension of plan-based theories of programming expertise.

3.5.8 Assessing claims about programming plans: Problems and limitations of the plan theory

The work of Soloway and his colleagues, and Rist's subsequent extension of plan-based programming theory, has clearly contributed to our understanding of the use of tacit knowledge in guiding problem solving in programming. In addition, this work has provided a useful means of describing the content of such knowledge structures and has been influential in the design of automated program analysis systems and intelligent tutoring aids. However, more recent work has questioned the basic adequacy of the plan theory. For instance, Pennington (1977 a and b) has conducted a number of studies (described below) which suggest that programmers form multiple representations from a program's text structure, and

that the nature of the representation that programmers actually develop is largely dependent upon features of the task that they are engaged in. Such representations include plan knowledge, but may also carry information relating to state and control flow. These studies question the centrality of the plan concept in program comprehension, but they do not provide sufficient evidence to reject plan theories. However more recently, a number of studies have questioned the basic theoretical claims that underlie plan-based approaches to program comprehension.

What claims do plan-based theories of program comprehension make?

Claim 1: Plans are language and problem independent

In order to assess the validity of plan-based theories of program comprehension it is necessary to make clear the theoretical claims that this approach advances. One of the major claims of plan-based theories is that programming plans will be both language and task/problem independent. Programming plans are taken to represent generic knowledge structures that in some sense represent the deep structure of a problem, hence a generic running total plan might be used in a variety of programs intended to solve a wide range of problems. In this way programs are seen to be constructed from these generic plans, which are then tailored according to the particular circumstances of a given problem. For example, the expert programmer wishing to compute facts about vehicle control will be able to access, say, a generic count plan , from their extensive repertoire of programming plans, and then instantiate this with variable names relevant to the particular situation, i.e., "count the vehicles". However a number of recent studies suggest the development of plan structures, or the ease with which plan structures might be comprehended or extracted from the program text, appears to be inextricably bound up with the way in which programming is taught, and this finding appears to pose serious implications for plan-based theories of programming expertise (see chapter 6).

The relationship between teaching and plan acquisition is problematic because if the acquisition of plan structures is the primary characteristic of programming expertise, as suggested by Soloway and Ehrlich, then differences in teaching strategy and educational background should not affect the nature of that expertise. Gilmore and Green (1988) highlight this problem with an analogy to instruction and expertise in chess:

"If the acquisition of plan structures is the defining quality of expertise, then differences in teaching strategy etc. should not affect the nature of expertise. For example, it is reasonable to argue that the nature of expertise in chess does not depend upon teaching strategies , because the nature of attack and defence and the configurations that represent them are inherent in the game. Similarly programming plans are assumed to be inherent in the problem, not in the language or the teaching, and the development of such schemata/plans is dependent on experience, not on teaching".

In chapter 6, experimental work is reported which suggests that the ability to extract plan structures from program texts is largely dependent upon the design experience possessed by the programmer, and that this design experience helps the programmer to construct a mapping between structures inthe problem domain (plans) and structures in the language domain (programs). Additional evidence for this idea is presented by Stone, Jordan and Wright, (1990) who provide experimental support for the idea that instruction in structured programming principles can improve debugging performance by increasing a programmer's comprehension of program goals and plans[4]. Taken together these findings appear to pose significant difficulties for plan theories of programming, since educational background and differences in teaching strategy per se should not affect the nature of plan knowledge or the acquisition of plans and their use.

Soloway and Ehrlich suggest that programming plans should be considered as schemata, however schema theory suggests that schemata acquisition is performed via an inductive process in which specific experiences are concatenated into generic schemata (Rumelhart and Norman, 1981). Schemata are not taught explicitly and we are certainly not taught procedures which facilitate the inspection and integration of schemata. However, while the work reported above does not suggest that plans are taught explicitly, it does indicate that differences in teaching strategy may influence the kinds of representations that programmers build and the way in which programmers use these generic representation.

Another problem with the plan theory is that all of the reported empirical work on program comprehension that has been used to provide support for plan-based theories has used programs written in a single language (usually Pascal or one of its close variants). However, the plan theory gains much of its theoretical force from the claim that plan structures are language independent. For instance

Soloway and Ehrlich claim that their experiments support the idea that "plan knowledge play(s) a powerful role in *program* comprehension" (p. 609) (my emphasis). However, their work was concerned only with the role of plans in the comprehension of Pascal or Pascal-like programs, the assumption presumably being that these effects will generalise to very different languages. Anderson (1985) claims that plans will be equally useful to expert Basic programmers as to expert Pascal programmers, but no evidence is cited for this. Indeed, until recently, there have in fact been few attempts to discover whether programmers using languages other than Pascal employ similar plans.

Gilmore and Green have recently called into question the generality of the programming plan as a description of the main type of cognitive representation employed by the expert programmer. Gilmore and Green (1988) have carried out a number of studies[5] which suggest that the notation of certain programming languages may make those languages amenable or otherwise to the identification and use of plans.This suggestion is based upon the finding that Basic programmers are unable to benefit from cues to plan structure (i.e., when such structures are colour highlighted) while debugging programs, while debugging success for Pascal programmers appears to be facilitated significantly by plan structure cues. Basic programmers appear not to employ and abstract plan-based representation of a particular program during program comprehension, but tend to rely more extensively upon control flow information implicit in the text structure of the program. Gilmore and Green claim that Basic is less "role-expressive" than other languages. That is, Basic programs are less discriminable from each other than are say Pascal programs. In the case of Pascal, they argue that features of the notation of the language, in particular its role expressiveness and lexical richness, make it easier for the programmer to infer the role of a particular statement and to discover the relationship between groups of statements.

This research calls into question the basic theoretical claim of plan theorists, which suggests that plan structures constitute a source of knowledge that is language independent. Indeed, if the plan theory is not generalisable to languages other than Pascal then clearly it is of significantly less utility and therefore interest. In chapter 5 a distinction is drawn which emphasises the different views which have emerged with respect to plan theories. One view, the traditional view of plans, suggests that programming plans are universal natural structures that characterise the expert programmer's mental representation of a program. In this sense such plans are thought to represent the deep structure of a programming problem.

Alternatively, we may claim that programming plans might best be regarded as artifacts of the particular notational properties that certain languages display.

Claim 2: The major defining characteristic of programming expertise is the possession of plans

The above discussion also has implications for the second major claim of the plan school: that is, that the possession of plan structures forms the basis for understanding programming expertise. The studies carried out by Gilmore and Green and by Davies that have been briefly reviewed above found that although their expert subjects displayed the same level of debugging performance, some of these subjects based their debugging strategies upon plan knowledge while others employed different forms of information. Hence in terms of debugging performance, programmer's behaviour may be equivocal, and it seems that plan knowledge, in this context at least, does not provide a cogent means of defining expertise. One of the main problems with the plan theory is that it seems that plans are used both to explain expertise and to provide the only empirical measure of that expertise. Note that in the studies carried out by Soloway and Ehrlich, no independent measures of expertise were established.

In chapter 7, I report a study which suggests that while the existence of plans may be used to differentiate novices from experts, an analysis of plan structures does not appear to provide a means of teasing out more subtle distinctions between levels of expertise. In particular, this study suggests that intermediate programmers appear to possess the same range and number of plan structures as their more experienced counterparts. However, intermediates and experts use plans rather differently. For example, while experts and intermediates are able to detect violations to plan structures with equal proficiency, intermediates take much longer to detect these violations in comparison to experts.

This finding suggests that the procedures for detecting plan violations may be compiled in the case of expert performance, thus leading to increased detection speed. Hence, the evidence concerning plan use seems to provide little support for the idea that the possession of plans can be taken as a defining characteristic of expertise. In terms of the plan theory, one would presumably expect plan structures to be gradually accumulated as a programmer becomes more experienced. However, the lack of a clear discontinuity in the number and range of

plans used by intermediates and novices suggests that this view is invalid. More importantly perhaps, this experimental work suggests that we need to consider not only a simple enumerative view of the relationship between plans and expertise, but that we also need to stress the importance of those factors that control plan use. This brings us to a consideration of the strategic aspects of programming behaviour which will be considered in more detail in chapter 4.

In the Architecture of Cognition (1983), John Anderson proposes a number of specific criticisms of schema theory that parallel some of the criticisms made here in response to plan-based theories of programming. Here, Anderson is concerned specifically with a comparison of the findings and the predictions of schema theory with his production system model of skill acquisition - ACT* (see chapter 2 for a detailed description of this model). He suggests that the major problem with schema theory is that it blurs the distinction between declarative and procedural knowledge and fails to explain the evident contrast between these two forms of knowledge. Anderson suggests that there are good reasons to have both declarative and procedural knowledge. Declarative knowledge is flexible and can be accessed in many ways, while procedural knowledge is rigid but efficient. Schemata, he suggests "are more declarative in nature and have similar flexibility" (pg 39). He goes on to suggest that :

"The condition-action asymmetry of production systems is committed to the idea that efficiency can be achieved by capitalizing on the structure and direction of information flow. One can only go from the instantiation of the condition to the execution of the action, not from the action to the condition. This contrasts with schemata ... where it is possible to instantiate any part and execute any other part. *The asymmetry of productions underlies the phenomenon that knowledge available in one situation may not be available in another. Schemata, with their equality of access for all components, cannot produce this*" (pg 39, my emphasis)

Another criticism that Anderson levels against schema theories is that such theories do not specify effective mechanisms for schema acquisition. He claims that technically, it is difficult to construct learning mechanisms that can deal with the full range of schema complexity. Anderson claims that "Empirically, it is transparent that learning is gradual and does not proceed in schema-sized jumps." (pg 39)

As we have seen these two criticisms of schema theory also apply to plan-based views of programming, and they serve to illustrate at least two of the central problems of the plan-based approach. The work reported later in this thesis attempts to extend and modify existing plan-based views of programming by suggesting a plan restructuring and control mechanism based upon experimental data concerned with plan comprehension and use. It will be suggested that this restructuring mechanism leads to asymmetrically structured schemata or plans which may have compiled elements, but in which certain salient plan structures remain accessible throughout the problem-solving activity.

3.6 Hybrid Models

Another apparent problem with much of the work that we have reviewed so far is that it considers knowledge representation in programming to be of a broadly uniform nature. For instance, analyses of programming behaviour based upon plan knowledge suggest that data flow and functional relationships will be central to program comprehension. However, this assumption has been challenged by Pennington (1987a) who carried out a series of experiments which suggest that programmers are able to form a number of diverse representations of a program text, and that the development of these representations appears to be broadly influenced by the demands of particular tasks.

Pennington was primarily interested in investigating whether procedural (control flow) or functional (plan knowledge) relations dominate programmers' mental representations of programs. She suggests that various types of knowledge about programming enable programmers to detect and represent the variety of relations that are implicit in a program text, and that the detection of these relations is a necessary condition of program understanding (Green, 1980; Green, Sime and Fitter, 1980; Pennington, 1985).

Pennington suggests that computer programs, like other forms of text, contain implicit information that must be detected in order to fully understand the program. For example, the sequence of statements in a program provides information about the sequence in which the program will be executed. Another type of information relates to the data flow of the program which is concerned with the changes or constancies in the meaning or value of program objects throughout the course of the programs execution. The notion that programmers abstract such information

from the implicit text structure of the program has close analogy to the idea that natural language texts are understood in terms of their underlying referential, causal, or logical relations (Kintsch, 1974; Meyer, 1975; Trabasso, Secco and van den Broeck, 1982).

Pennington suggests that four main types of abstraction can take place. The first abstraction involves inferring a program's goals , i.e, what the program is supposed to accomplish or produce. Pennington suggests that this abstraction is of a functional nature and will contain little explicit information about how these goals will actually be accomplished. A second abstraction involves extracting information from the text about processes that transform the initial data objects into the outputs of the program, i.e., a data flow abstraction. A third abstraction might involve the production of a control flow representation, indicating the passage of execution control. A final abstraction, may involve extracting information about the program action that will result when a particular set of conditions is true. This abstraction resembles a decision table in which each possible state of the world is associated with its consequences.

In her experimental studies, Pennington provided a number of different types of comprehension question. Each type of question was phrased such that it accessed a different type of information from the program (e.g., Will an average be computed? (function); is the last record in ORDER_FILE counted in COUNT_CLIENTS? (sequence); will the value of COUNT_CLIENTS affect the value of ACTIVE_AVG? (data flow); When TEXT_EXIT is reached, will ORDER_REC_ID have a particular known value? (state)). Subjects were given a program to study and were told that they would be asked to respond to comprehension questions and be given a subsequent memory test. Following their study of the program text, subjects were asked to respond to each of the comprehension questions. They then carried out a free recall task where they were asked to recall as much of the program as possible, in whatever order occurred to them. Finally, the subjects participated in a complex priming task in which the time taken to recognise an individual line from the program was recorded.

Pennington was interested in exploring a number of hypotheses. However, her specific concern addressed the hypothesis that if programmers form plan-based representations of programs, then they should recognise lines faster when they are preceded by lines derived from the same plan structure. This hypothesis is based upon the assumption that activation of an item in the memory structure will activate

items close to it, and especially those in the same cognitive unit. Hence, response time to the target preceded by an item in the same cognitive unit should be faster than the response time to an item not in the same cognitive unit (Anderson, 1983; McKoon and Ratcliff, 1980); that is, a priming effect should be apparent.

Pennington's results suggest a number of problems with plan-based theories of programming expertise. Firstly, her subjects made fewer errors on the control-flow questions compared to the data-flow and the function questions. Moreover, the effect of priming was greater when the primes were derived from the same control structure as opposed to being derived from the same plan structure. Hence, it appears that representations of control flow appear to dominate a programmer's representation of a program, and that representations of function and data-flow, which are more naturally allied to plans, are not as important as a basis for organising memory structures.

Pennington replicated these findings in a second study to which an additional stage was added. In this second experiment, her subjects were required to make modifications to a program and half were asked to provide verbal protocols while carrying out this task. Pennington discovered that after the modification phase the dominant representations were concerned with data-flow and function, and this was especially true for those who had supplied protocols. These results suggest that a programmer's task goals may also influence the structural relations that dominate mental representations in comprehension

Pennington concludes by stating that her results "strongly support a view of program comprehension in which abstract knowledge of the program text plays the *initial* organising role in memory for programs, and that control flow or procedural relations dominate in the macrostructure memory representation" (p. 337). She further claims that these results are consistent with the results found in prose text comprehension which suggest that knowledge of narrative and expository text structures guides comprehension and plays an important role above and beyond other content-schematic factors (e.g., Cirilo and Foss, 1980; Haberlandt, Berian and Sandson, 1980; Mandler and Johnson, 1977).

Pennington's work suggests that knowledge representation in programming takes on a hybrid form with various sources of knowledge contributing to the development of memory organisation. Hence, certain forms of knowledge may play an important organising role during different stages of comprehension.

Moreover, it appears that the kinds of tasks undertaken by programmers may influence the nature of the mental representation of a program. This view clearly does not rule out a role for plan-based knowledge structures in program comprehension. It does however, challenge their centrality as structures that are hypothesised as being fundamental to program comprehension. Hence as Pennington suggests "While plan knowledge may well be implicated in some phases of understanding and answering questions about programs, the relations embodied in the proposed plans do not appear to form the organising principles for memory structures." (p. 327).

3.7 Conclusions

This chapter has served to provide a broad review of the extensive literature concerned with knowledge representation in programming. To a large extent the focus of work in psychological studies of programming has paralleled the concerns expressed in other problem-solving areas. For instance, early studies of programming concentrated upon the general structural aspects of knowledge representation, and provided important supplementary support for the chunking hypothesis that originated in other problem-solving domains such as chess and physics. Later studies began to focus more explicitly upon a description of the content of expert and novice knowledge structures. The concern of these later studies appears to mirror the trend apparent in other problem-solving domains, where interest began to be directed toward understanding the nature of generic knowledge structures and their role in problem-solving and comprehension. This is perhaps best exemplified by the work of Schank and his colleagues in their development of schema theory and the subsequent application of this theory to program comprehension by Soloway and others.

More recently, the centrality of the plan concept has been challenged by studies which suggest that programmers can extract many different forms of programming knowledge from a given text structure. In addition, the studies reported later in this thesis, and outlined briefly here, highlight some major theoretical difficulties with plan-based theories of programming expertise and program comprehension. It appears that there is still much scope for further research into knowledge representation in programming. For example, we know little about the mechanisms that are involved in plan acquisition. In addition, while plan-based theories provide a detailed description of programming knowledge, they have little

to contribute to our understanding of the way in which plans are used. One intention of this thesis is to provide a general theoretical framework within which these issues can be addressed.

While this chapter has sought to suggest that studies of programming expertise have broadly paralleled studies of expertise in other domains, it should be noted that some of the outstanding features of expertise that have been observed in non-programming domains have not found a place in general accounts of programming behaviour. For instance, most descriptions of knowledge representation in programming appear to suggest that such representations are uniform and that all parts of the representation are equally accessible. However, there is substantial evidence to suggest that the development of expertise is associated with changes in the way in which knowledge is represented.

For instance, it has been observed that a significant part of the development of skill in a particular domain involves the transformation of declarative knowledge into procedural knowledge (Anderson, 1982; 1983; 1987; Neves and Anderson, 1981; and see review in chapter 2). However, most theories of the development of programming skill do not make a distinction between these two forms of domain knowledge. For instance, the plan theory of programming appears to suggest that the knowledge expressed by plans has both declarative and procedural elements. Hence, a plan may specify the actions necessary to compute a particular procedure or it may describe, in a stereotypic fashion, the contents of that procedure. Characterising programming knowledge in this way clearly does not lead to a theoretical explanation of skill development in programming that is congruent with alternative accounts of skill development in similarly complex domains.

A fundamental aspect of the framework suggested by this thesis is that as skill develops knowledge may be both proceduralised and/or restructured. Hence, it is suggested that certain key elements of plans remain accessible even though other elements of the plan may be compiled and proceduralised. The experiments reported in this thesis provide support for the idea that certain salient elements of plans are used to guide skilled problem solving behaviour in programming. For instance, in a study of program generation reported in chapter 8, expert programmers were observed to develop their programs by instantiating a focal plan element and then accreting the rest of the plan around that focal element. In chapter 10, an experiment is reported which demonstrates that expert programmers can access these focal plan elements both more quickly and more accurately than

novices or intermediates. In addition, a third experiment (reported in chapter 7) has shown that while experts and intermediates can detect plan violations with about the same frequency, expert programmers detect such violations with greater rapidity than intermediates.

These findings are interpreted within a general framework which suggests that a critical factor in the development of expertise in programming is not simply related to the development of a larger catalogue of plan knowledge, as suggested by the plan theorists. Rather, expertise appears to be related to the way in which plan knowledge is structured. Similarly, in other domains, such knowledge restructuring processes have been proposed as central elements in theories of skill acquisition (for instance, Kay and Black's (1984) work on text editing and Lewis's (1981) work on algebra). In addition, Cheng (1985) has taken these ideas further by suggesting that the enhanced performance exhibited by experts in certain domains, and accounted for by classical problem solving theories in terms of knowledge compilation and proceduralization, can be equally well interpreted by proposing a knowledge restructuring mechanism which can restructure task components so that they are coordinated, integrated or reorganised into new perceptual or cognitive units.

This thesis attempts to relate this knowledge restructuring process (elaborated in the final chapter of the thesis) to a subject's exposure to design experience and to the way in which this experience may impact upon a programmer's knowledge organisation in relation to programming tasks. In addition, an attempt is made to illustrate how this knowledge restructuring process, in concert with the effects of certain notational features, can give rise to particular forms of strategy. This leads to the development of a model of programming behaviour which suggests possible ways in which external task and language features might interact with a programmers internal representation of programming knowledge. It will be suggested that this model accounts parsimoniously for existing experimental data and that it is not open to the kind of criticism that is typically levelled at plan-based theories of programming expertise.

Notes

[1] It should be noted that this analogy considers programs as communicative entities. However the analogy loses strength if one considers the dual nature of programs: that is, they can be executed for effect as well as being read as communicative entities.

[2] See Kernighan and Plauger (1978) for a collection of rules derived from actual programming practice. Soloway and Ehrlich equate these rules with their notation of program discourse rules.

[3] Note that the Soloway et al study used programs where the experimenters had deliberately violated certain discourse rules. It is likely that it would be more difficult to assess accuratly the extent to which arbitarily generated programs violate certain rules in more naturalistic contexts.

[4] Readers are refered to chapter 6 for a more detailed description of these studies

5 Readers are refered to chapter 5 for a more detailed description of these studies

Chapter 4. Studies of the strategic aspects of programming skill

4.1 Introduction

The previous chapter addressed issues stemming from what we might refer to as knowledge-based theories of programming expertise - that is, theories of expertise that have been concerned with the content and structure of programming knowledge. However, as we have seen, one major limitation of many of these knowledge-based theories is that they often fail to consider the way in which knowledge is used or applied. Hence, such theories have tended to concentrate simply upon the description or characterisation of declarative and/or procedural knowledge structures with the primary intention of demonstrating novice/expert differences in knowledge representation. However, such theories have failed to elaborate the cognitive mechanisms that may be thought to underpin the utilisation of such knowledge and have, by and large, ignored the strategic elements of expertise which are likely to play an important role in any comprehensive theory of programming skill.

To take one example, proponents of the plan theory of programming have devoted considerable effort to providing a detailed description of the kinds of knowledge structures that are thought to underlie expertise. However, the plan theory does not suggest how plans might be used. Hence, it might be claimed that the plan theory simply presents a theory of plans rather than a theory of planning.

The main assumption of plan-based theories is that the cognitive processes underlying programming are relatively straightforward (Gilmore 1990). Based upon ideas from artificial intelligence, it is suggested that these processes should be considered to be general problem-solving skills (cf. ACT*; Anderson (1983), and SOAR; Laird, Newell and Rosenbloom, 1987), which are possessed not only by experts but also by novices. Hence, the development of expertise is seen to be associated with the gradual accumulation of plan knowledge over time, rather than with the development of processes or heuristics which may govern plan use and application. Kolodner (1983) encapsulates the main problem with

knowledge-based theories of expertise: "even if a novice and an expert had the same semantic knowledge..., the expert's experience would have allowed him (sic) to build up better episodic definitions of how to use it".

An alternative to knowledge-based theories is to suggest that expertise in programming may involve the development or acquisition of complex task-specific cognitive or problem-solving strategies (Shneiderman and Mayer, 1979). Two interpretations of this view are possible. One interpretation, may be to suggest that the development of expertise involves the acquisition of strategies rather than declarative knowledge. This view suggests that even if we were able to teach expert knowledge structures to novices, this would not make them into experts since they will not have acquired strategies for utilising this knowledge (See Neisser (1976) for a more general discussion of this issue in the context of other problem solving skills).

Alternatively, one may adopt a less strict position and suggest that expertise has both knowledge-based and strategic components. One might for instance argue that features of a programmer's knowledge representation determine the form of strategy that they adopt. This is the position suggested by the work reported in this thesis where an attempt is made to demonstrate the relationship between the development of expertise and the adoption of particular forms of strategy. Moreover, in contrast with existing work, this thesis is not simply concerned with characterising the forms of strategy that are seen to be associated with particular levels of skill.

Rather, it is concerned with explaining why these strategies emerge. In particular, attention is directed toward exploring the relationship between the development of structured representations of programming knowledge and the adoption of specific forms of strategy. It will be suggested (see the final chapter of this thesis) that a knowlege restructuring process occurs during the acqusition of expertise which results in the differential accessibility of various knowledge structures. Moreover, it is hypothesised that this gives rise to different forms of strategy. Hence, the work reported in this thesis represents an attempt to adopt a rather different perspective on the relationship between expertise and the emergence of particular forms of programming strategy. This approach provides an explanation for the

differences in strategy that have been observed in studies of programming behaviour and attempts to relate these strategic differences to an explicit model of knowledge representation in programming.

This chapter considers a number of studies which have been concerned with the role of strategy in programming. These studies have tended to explore either the strategic aspects of program comprehension or the role of strategy in program generation. A number of studies have been concerned only with the strategies employed by experts, while others have attempted to associate differences in strategy with different levels of expertise. A number of other studies have been explicitly concerned with the kinds of difficulties experienced by novices, and in particular with those difficulties that arise because of an absence of elementary problem-solving strategies or because of a reliance upon inappropriate strategies. This chapter attempts to represent these different concerns and is composed of three sections. The first section deals with studies of program comprehension strategy, while the second reviews studies of generation strategy. A third section deals more explicitly with studies of novices, and attempts to highlight the fact that novices can display strategic as opposed to, or in addition to, knowledge-based programming difficulties.

4.2 Strategies involved in program comprehension

4.2.1 Brook's model of program comprehension

One of the earliest theoretical explorations of the strategic aspects of program comprehension was outlined by Brooks (1977; 1983). Brooks presents what he calls a sufficiency theory which is intended as a description of the processes by which a programmer attempts to understand a program. A sufficiency theory, according to Brooks, should provide a description of a set of mechanisms and relationships that are sufficient to explain at least the most salient aspects of program comprehension behaviour. Brooks presents a number of behavioral or empirical differences that he claims any model of program comprehension should be able to account for. These sources of variation include the following:

1. The effects of differences in the task performed by the program on comprehension. Why do programs that perform different computations differ in comprehensibility, even though the intrinsic properties of these programs (e.g., length, complexity etc) are the same?

2. The effects of variation in program text. Why, and under what conditions, do programs written in different languages vary in comprehensibility even though they perform the same computation?

3. The effects of different programming tasks. Why might the comprehension process differ depending upon the nature of the task the programmer is undertaking. For instance, modifying a program as opposed to debugging it?

4. The effects of individual differences. Why might one programmer find a particular program easier to comprehend than another programmer?

Brooks then goes on to propose a model to explain these sources of variation. Brooks claims that the mechanisms suggested by this model are sufficient to explain the variability in the extent to which particular programs can be understood. He suggests that in this respect the model takes the form of a theory demonstration approach (Miller, 1978).

Brooks outlines three main elements of his model of program comprehension in terms of a set of domain mappings and processes. He suggests that the programming process involves constructing mappings from a problem domain into the programming domain. This mapping process may also involve a number of intermediate domains . To illustrate this mapping process he considers a cargo-routing problem. Here, objects in the problem domain are cargoes that have specific destinations which must be reached within particular time and cost constraints. Before a program can be constructed to solve a routing problem, the programmer must assign numbers to the various cost and time elements, and identifiers (which might also be numbers) to the cargoes and destinations. This results in what Brooks refers to as a new knowledge domain, where the domain objects have become numbers. Next, an algorithm must be selected to carry out

the required computation. This gives rise to a further domain in which mathematical objects, such as trees or arrays, are constructed and in which various operations are specified, such as matrix inversion. Yet another domain emerges when the program itself is constructed and these mathematical objects are implemented as data structures and as primitive programming language operations. Finally, the execution of the program results in a further domain in which objects become the contents of memory locations and operations are implemented as low-level machine code instructions.

Brooks claims that the task of understanding programs requires programmers to construct or to reconstruct enough salient information about these domains in order to provide a basis for generating various mappings between the domains. This construction or reconstruction process involves the programmer acquiring two main forms of information. One form of information is contained within each domain. That is, information about the basic set of objects in the domain and their relationships. The second form of information relates to the relationships between objects and operators in one domain and those in nearby domains.

Brooks suggests that this construction/reconstruction process is expectation driven and consists of confirming/refining various hypotheses which are generated from the programmer's knowledge of the task domain and other related domains. According to Brooks, this process begins with a primary hypothesis which is generated when the programmer obtains any information about the task that the program performs. This primary hypothesis specifies the global structure of the program in terms of its inputs, outputs, data structures and processing sequences.

The next stage in Brooks' model involves verifying or validating this primary hypothesis. This verification process will normally involve finding evidence for the hypothesis in the program code or in its associated documentation. Brooks further claims that since the primary hypothesis will almost always be global and non-specific, the programmer will have to generate a number of subsidiary, and less detailed, hypotheses which can be verified against information obtained from the code and the documentation. Brooks claims that these subsidiary hypotheses can be regarded as forming a hierarchical structure where those hypotheses lower in the hierarchy represent specialisations of those occurring above.

Brooks asserts that to minimise memory load, these subsidiary hypotheses will normally be created in a top-down, depth first manner. Hence, the comprehension process begins with the creation of a primary, and then a number of subsidiary hypotheses about the program's function. Initially these are based upon the programmers knowledge of the domain and of similar programs. Eventually, these subsidiary hypotheses become sufficiently detailed to enable the programmer to directly verify them against program text or documentation. The success or failure of this verification process can then be used to guide the formation or the modification of other subsidiary hypotheses. Hence, the condition that causes the hypothesis refinement process to terminate occurs when the level of detail of a hypothesis is sufficiently close to the program text or documentation to enable a specific comparison. More specifically, this process terminates when the operations or data structures specified in the hypothesis are ones that the programmer can associate with features or details visible in the program text which are typical indicators of the particular operation or structure in question.

Brooks refers to the features that typically indicate the occurrence of certain structures or operations within the code as 'beacons'. For example, a typical indicator for a procedure that sorts array elements might be a section of code in which the values of an array element are interchanged. Clearly, there might be multiple beacons for a single structure or conversely the same beacon might represent a variety of structures or operations.

In summary, Brooks' theory of program comprehension holds that comprehension is a top-down, hypothesis driven activity. According to this theory, the programmer does not study a program line-by-line, but instead forms hypotheses about program function based upon high-level domain and programming knowledge. These hypotheses, once suitably decomposed, are then verified against the program text by searching for beacons which indicate the presence of particular functions or structures. A particular hypothesis is verified when the expected beacons are found. If these are not found, the program text might be searched more thoroughly. If this more detailed search fails, the hypothesis is weakened and will be revised or discarded. One major difference between Brooks' theory and the studies described in the previous chapter is that this theory recogises the more global level of program structures and suggests, in

turn, an interaction between knowledge structures and information that is derived more directly from the program.

Brooks claims that his theory of program comprehension can account for the sources of variation in program comprehension outlined earlier. For instance, there is considerable evidence that the type of control structures used to express programs can have a marked effect upon their comprehensibility (Green, 1977; Soloway, Ehrlich, Bonar and Greenspan, 1982). Hence, the text structure of the program may contribute to the ease with which hypotheses can be verified. An even more powerful source of variation may be related to the tasks the program is intended to perform. Hence, if as suggested by Brooks, program comprehension involves reconstructing the relationship between the original problem and the program text, then ease of comprehension will depend to a large extent upon the complexity of the original problem. In support of this, Brooks cites a study by Curtis, Sheppard, Milliman, Borst and Love, (1979) which showed significant differences in the comprehensibility of programs with the same software metric[1] value.

The theory also suggests at least three distinct factors that might contribute to individual differences in program comprehension ability - programming knowledge, domain knowledge and comprehension strategies. Firstly, a programmer's ability to confirm hypothesis against code or to refine hypotheses appropriately will depend to a certain extent upon the programmer's knowledge of typical programming idioms and algorithms.

Secondly, since Brooks' theory suggests that domain knowledge is critical to the formulation of high-level hypotheses, one might predict that a programmer will have great difficulty generating useful hypotheses if that programmer does not understand the problem the program is intended to solve. To the author's knowledge this prediction has not been tested; however it does suggest that documenting the rationale behind the specification for a program may aid comprehension by illustrating salient features of the problem domain.

Finally, interprogrammer variation may arise from differences in the strategies employed by programmers to locate information in the program text. For instance,

two programmers attempting to validate the same hypothesis may use different strategies. One programmer may attempt to validate it by tracing the subroutine calling hierarchy, while another might attempt to locate input and output functions. Brooks claims that while these strategy differences may not account for as much individual programmer variation as programming and domain knowledge, they may play a role in explaining pathological cases of programmers who are exceptionally successful or unsuccessful at program comprehension.

4.2.2 Empirical support for the Brooks model

Brooks' theory of program comprehension provides a detailed and wideranging account of the strategies thought to be employed by programmers when they attempt to understand a program. However, while Brooks cites existing experimental work in support of the theory, no specific empirical evidence for the model is proposed. More recently a number of studies have been undertaken which address specific hypotheses derived from Brooks' model. For instance, Wiedenbeck (1986a and b) and Wiedenbeck and Scholtz (1989) have provided evidence for the idea that programmers use beacons to guide their problem-solving behaviour during program comprehension.

The role of beacons in program comprehension

For instance, in one of these studies (Wiedenbeck and Scholtz, 1989) a comparison was made between the comprehension of programs containing beacons and other similar programs which did not contain a beacon. Wiedenbeck and Scholtz constructed two versions of a Shellsort program, one containing a beacon line which swapped various values in a standard manner (i.e., t := a[j]; a[j] := a[j + i]; a[j + i] := t;). A second 'disguised' version of the program introduced a whole second array, which was a copy of the first and served as a temporary storage for the swap value.The swap was performed by assigning the values to be swapped into the appropriate locations of this second array, then later copying the second array back into the first. Wiedenbeck and Scholtz claim that this second version is almost identical to the first, but that in the second case the beacon is disguised.

Novice and advanced subjects were given the programs to study for a short period and were then asked to carry out three tasks. The first task involved describing the function of the program. The second task involved subjects judging their confidence in having understood the program and a third task required subjects to recall the program verbatim. The first task was intended to demonstrate the subjects' understanding of the program and the confidence rating was used to provide a supplementary comprehension measure, since subjects might perform as well on the disguised version, but be less sure of themselves in respect to their understanding of the program. The recall measure was introduced to determine whether subjects remembered the swap better when it was presented in its more typical form.

Wiedenbeck and Scholtz found that the advanced group was better at determining program function than the novice group and that, overall, subjects were more accurate in determining the function of the normal (no-disguise) version. They argue that these differences appear to arise as a consequence of the superior comprehension performance of the advanced subjects on the no-disguise version. Similar findings emerged in the case of the confidence rating task. Here advanced subjects were significantly more confident of their function judgments than were novices, and confidence was higher for function judgments in the no-disguise version than in the disguise version. In the recall task, there was a trend for subjects to recall the swap lines in the no-disguise version better than in the disguise version (67% vs 40% correct recall); however this difference was not significant.

This experiment provides some evidence for the role of beacons in program comprehension and suggests that at least part of Brooks' model maybe correct. The idea that program comprehension behaviour is guided by beacons which serve to highlight important code structures is central to the model of program generation presented later in this thesis. The model proposed in this thesis places emphasis upon the process of knowledge restructuring that appears to be associated with the development of expertise. The model proposes that certain focal elements of plan-based knowledge structures will be more easily accessible and that these focal elements will tend to be generated first during program development. Subsequently, these focal structures will be further expanded to form a complete

solution. Correspondingly, we might consider program beacons to form an external analogue of these internally represented focal structures. Hence, the ease with which particular languages support the expression of beacons is likely to affect comprehension success.

Strategic differences in program comprehension

While the studies reported above by Wiedenbeck and her colleagues have served to provide a direct test of one aspect of Brooks' theory, other work has indirectly addressed issues stemming from the ideas on program comprehension that are presented by Brooks. For instance, Pennington (1987b) has carried out a study looking at the way in which differences in comprehension strategy can affect the level of comprehension achieved by expert programmers. Pennington carried out a detailed protocol analysis of the verbalisations of 40 professional programmers who were asked to study and modify a program. After a 45 minute study period her subjects were asked to summarise the program they had studied and respond to a number of comprehension questions.

Pennington analysed her subjects' verbal protocols by classifying each statement in the program summary according to two explicit dimensions. Firstly, statements were classified according to their type, i.e., as a procedural, a data flow or a function statement. Secondly, each statement was classified in terms of its referent. For instance, a statement might refer to specific program operations or variables. Pennington classifies such statements as *detailed* statements. Alternatively, statements might refer to a program's procedural elements, and Pennington calls these *program* level statements. *Domain* level statements refer to real world objects such as cables or buildings and, finally, statements with no explicit referent were classified as *vague* statements.

The program comprehension questions were used to classify the subject population into an upper (Q1) and a lower (Q4) comprehension quartile. Pennington then looked at the differences between these two groups in terms of the protocol classification scheme described above. With respect to the statement type classification, Pennington found no reliable differences between the Q1 and

Q4 groups. However, analysis of the referent classification illustrated a number of important differences between the upper and lower quartile comprehenders. Specifically, the Q1 group produced fewer statements at both the detailed level and at the vague level.

To explore the Q1 and Q4 differences further, Pennington sorted the program summaries produced by her subjects into three different "summary strategy" groups according to the proportion of statements at different levels in each summary. The first group (nine subjects) was composed of those programmers whose summaries predominantly consisted of statements corresponding to the program level. These summaries were referred to as program level summaries. The second group (twenty subjects) showed a more even distribution over program and domain levels, and their summaries were referred to as cross-referenced summaries. Finally, a third group (eleven subjects) produced summaries that included a high proportion of domain or vague statements and these were called domain summaries.

Pennington suggests that to construct program summaries, subjects must retrieve information from one or more memory representations and that consequently we may assume that these summaries reflect at least some properties of the subject's mental representation[2]. Pennington claims that abstract knowledge of a program text structure plays an initial organising role in memory for programs, and at this stage tends to dominate the macrostructure (van Dijk and Kintsch, 1983) memory representation that Pennington refers to as a program model. A second representation is created at later stages in program comprehension that reflects the functional structure of the program and is expressed in the language of the real world domain to which the program is applied. This Pennington calls the domain model.

It appears that Pennington's work supports the idea initially proposed by Brooks that a significant factor in program comprehension involves creating a successful mapping between the problem and the programming domain - in terms of Pennington's study, between the domain model and the program model. One can identify three distinct strategies employed by Pennington's subjects. One strategy, the program level strategy, is characterised by the almost complete absence of

order in which the program was written and were expected to be able to describe how the program worked. For both of these groups reading times and search patterns were collected. A third group of subjects viewed a coherent program, but were asked to talk about what they were doing as they inspected the program. In this case, only verbal protocols and search patterns were analysed.

The reading time data and search strategy information collected from the first two experimental groups illustrated the varied nature of cognitive processes during code comprehension. Reading time data indicated that subjects who viewed a coherent program spent an average of around 50 minutes studying the program lines, while those reading a scrambled program spent an average of only 6.5 minutes reading the program. Almost all of the subjects who viewed the coherent program made short 'retrogressions' through the code. In order to analyse this phenomenon further, Robertson et al, categorised each action performed by the subjects as either a forward move (from one line to the next after another forward move), backward move (from one line to the previous line after another backward move) or as a switch in direction. Switches could in turn be categorised as either forward-backward switches (a return to a previous line after a forward move), or as backward-forward switches (a return to a subsequent line after a backward move). The data relating to these switching episodes indicates that around 11% of the subjects activity involved switches in direction, of this 16.6% involved going backwards through the code and the remaining 72.4% constituted forward movement.

Search patterns were characterised by segmenting each subject's protocol into episodes. An episode consisted of a forward pass through a section of code, a forward-backward switch, and a second forward pass. All of the subjects' data contained a number of such episodes, and about one third of subjects' activities were categorised as being within episodes. Next, Robertson et al analysed reading times for the various movement types (i.e, forward, backward, forward-backward, backward-forward etc) and for between and within episode activities. These data are shown in table 4.1.

There was a significant difference between the reading times within episodes and between episodes, and Robertson et al take this to imply that the

"episode/non-episode distinction may be due to the highly goal directed nature of processing within episodes" (p. 960). They suggest that between episode processes are likely to include more discovery based processing, whereas during within episode activity, subjects are searching for particular information. We might here draw parallels with Brooks' model and suggest that between episode processing may give rise to hypothesis generation, while within episode processing may provide a mechanism for hypothesis verification. Although Robertson et al do not mention Brooks' model, it does appear that their data provides support for the comprehension strategies that Brooks alludes to in the context of his model.

Movement Type	Between Episodes	Within Episodes
Forward	408	---
Backward	141	146
Forward-Backward	1581	1027
Backward-Forward	569	410
First Forward Pass	---	387
Second Forward Pass	---	225
Episode begin	---	348
Episode end	---	411

Figure 4.1. Reading times for the various movement types reported by Robertson et al (1990)

The protocol analysis carried out by Robertson et al also suggests that certain movement types are associated with particular kinds of problem-solving activity. Here, the programmers' verbal comments were classified into six groups: Analyse, assume, question, answer, function and strategy. These were defined as follows:

Analyse - comments which a programmer offered as an explanation of a code segment.

Assume - comments where a programmer made prediction about what she expected next in the program.

Question - a comment that contained a query about the code

Answer - a comment or statement that could be explicitly linked to an earlier question.

Function - a comment about the function of a particular piece of code.

Strategy - a comment about what the programmer planned to do next.

Table 4.2 shows the proportion of comment types within each move category. It appears that concern for the functionality of the code was the topic of most of the programmers comments. This concern for functionality was apparent for all movement types with the exception of the backward movement category. In the case of this movement category, questions and strategy were the primary concerns. Robertson et al suggest that "the unequal distribution of comment types across movement categories shows that programmers had qualitatively different things in mind when they moved around in the code" (p. 963)

Comment Category	Movement Type			
	Forward	Backward	Forward Backward	Backward Forward
Analyse	.078	.053	0	.057
Assume	.219	.158	.167	.220
Question	.078	.263	.125	.118
Answer	.094	.105	.083	.118
Function	.422	.158	.250	.368
Strategy	.109	.263	.375	.103

Figure 4.2. Proportion of movement types in eaxh movement category.

These data might be seen as providing support for other models of program comprehension. For instance, Brooks' model places emphasis upon the idea that programmers generate and test hypotheses about the function and role of elements of code when they attempt to understand a program. However, the nature of these hypotheses remains unexplored. The contribution of the Robertson et al study is that it provides a detailed analysis of the kinds of questions that are framed by programmers during comprehension. This analysis enables us to state more clearly the detailed nature of the hypotheses that programmers generate In addition, it provides a basis for a more detailed understanding the nature of programmers' information seeking activities which are directed towards testing these hypotheses and which are are evident in the strategies that they employ during comprehension.

Widowski and Eyferth (1986) have conducted a study of programmers' searching strategies using a methodology similar to that employed by Robertson et al. Widowski and Eyferth were interested in comparing the strategies used by novice

and expert programmers who were asked to study a program line-by-line through a manoeuvrable, single line window. Widowski and Eyferth extended the traditional application of the Chase and Simon (1973) chess research which compared the performance of novices and experts on well-ordered versus randomly-ordered game positions. In the programming domain, the effects of well ordered vs random program presentation are well documented (see chapter 3), and studies of this effect in programming have replicated the basic finding of Chase and Simon who showed that expert performance is drastically reduced in the random condition, while novice performance is affected little. This effect is thought to be based upon the expert's ability to extract meaningful chunks from the materials presented, be they chess positions or programs.

Widowski and Eyferth suggest that the same effect should obtain in situations where programs are well ordered, but which are unconventional or atypical. This would be predicted by the Chase and Simon model, since, although a program may be well ordered, it may not always be possible for a programmer to decompose it into meaningful chunks. In the case of an atypical program, programmers may not posses sthe appropriate knowledge structures upon which to base their chunking strategies. Hence, in the case of atypical (or semantically complex) programs, we might expect novice-expert differences to diminish, since experts may not have the knowledge structures necessary to process them.

Widowski and Eyferth compared the performance of a number of Pascal novices and experts who were allowed to view only a single line of a program at a time. For the purpose of their experiment, Widowski and Eyferth constructed two pairs of programs which differed in their level of semantic complexity or typicality. Each pair of programs consisted of a stereotypical (semantically simple) and a non-stereotypical (semantically complex) program. Each pair of programs was the same length and the programs had an equivalent number of variables. Subjects were allowed to study each program for 10 minutes.

In a subsequent recall study, significant effects of expertise and semantic complexity were evident. However, there was no interaction between expertise and complexity. Contrary to the hypothesis derived from Chase and Simon's model, experts were much better than novices at recalling both typical and atypical

programs. Indeed, the difference in novice and expert performance was, if anything, greater in the atypical condition.

Next, Widowski and Eyferth analysed the reading patterns of their subjects in order to derive information on the search strategies they used. The basic form of measurement used here involved analysing the number of "runs" made by subjects. Here, a "run" was evidenced by the subject reading a number of consecutive lines before moving either backwards or forwards over more than one line in order to read another line. They suggest that this simple unit can reveal parts of the program text "that form subjectively meaningful entities" (pg 273). Widowski and Eyferth found that experts demonstrated more flexible reading strategies that novices. In the case of the stereotypical programs, experts adopted a strategy that involved reading the code in long but infrequent runs. In the case of the atypical programs, experts read the program in short and frequent runs. They suggest that the former strategy reflects a top-down or conceptually driven comprehension process, while the latter represents a bottom-up and heuristically oriented strategy. For novices, the same comprehension strategy appeared to be employed for both the typical and atypical programs.

The contribution of this study to our understanding of program comprehension is twofold. On the one hand, this study demonstrates that experts have a better ability to respond to novel situations, even though they may not have the appropriate knowledge structures or plans to guide behaviour. Secondly experts appear to have a greater range of strategies available to them, leading to greater flexibility in performance. Hence, this work provides an interesting counterpoint to plan-based theories of programming expertise by suggesting the important role played by strategy in program comprehension.

The studies conducted by Robertson et al and by Widowski and Eyferth have demonstrated the importance of the strategic elements of program comprehension. However, in order to achieve this, and to render their studies possible, they have restricted the task environment to such an extent that one might wish to draw into question the ecological validity of their experiments. One way of studying comprehension strategies which does not involve undue restriction to the task environment might be to examine the eye movement and fixation patterns made by

programmers while reading programs for comprehension purposes. However, studies of program comprehension which have involved eye movement analyses are rare.

One study of subject differences in program reading has been conducted by Crosby and Stelovsky (1989). They asked expert and novice programmers to study a short binary search program. The subjects were asked to study the program for as long as they needed to understand it and a record of the total number of fixations made by the subjects and their associated durations were collected.

Figure 4.3 illustrates two rather distinct reading patterns that emerged in Crosby and Stelovsky's study. These diagrams show the eye fixation patterns of two subjects superimposed over the algorithm. Each fixation point is represented here as a circle, whose size is propotional to the size of the pupil. Figure 4.3 a illustrates a typical "left-to-right, top-to-bottom reading strategy", of the kind that might be found in studies of nonprocedural text-based prose (Just and Carpenter, 1980). A rather different reading style is shown in Figure 4.3 b. This subject first concentrated on the upper three lines of the algorithm, then skipped a number of lines to concentrate on the central portion of the program. Notice that this subject repetedly fixated upon certain areas of the algorithm. Crosby and Stelovksy suggest that "rereading was not only a frequent practice, but a necessity for comprenension. If an internal buffer held the information (as, for instance suggested by Bouma and de Voogd's (1974) model of reading), frequent rereading would be superfluous" (p. 141).

Crosby and Stelovsky provide a more detailed analysis of individual scanning strategies by producing two dimensional "area/fixation" graphs which indicate the sequence of areas perused with respect to time. They found that patterns of eye fixations could be categorised, although there was a considerable range of individual scanning strategies. Such strategies ranged from linear scans of the text, typical of the kind found in prose reading, to highly nonlinear strategies. One might expect these differences in strategy to be related in some way to the expertise of the programmer, since the nonlinear strategy would appear to be highly goal directed, whereas the linear strategy treats each part the program

equally. However, Crosby and Stelovsky found no relationship between the use of particular strategies and expertise.

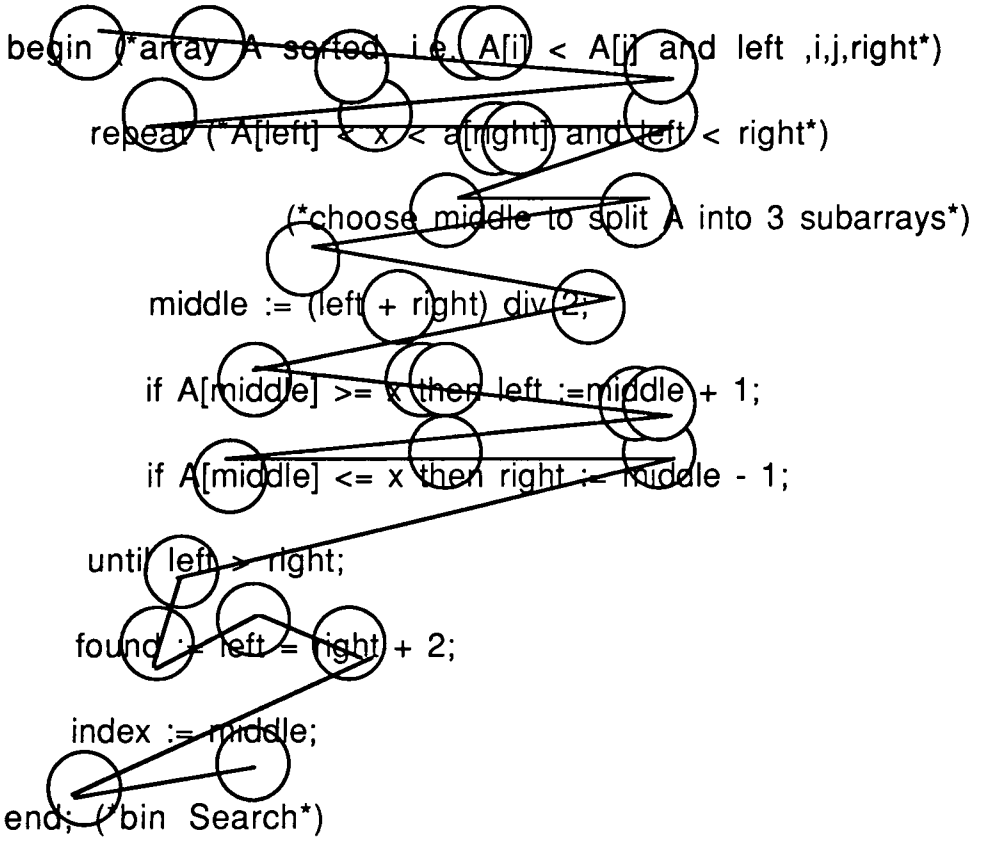


Figure 4.3a. A typical left-to-right, top-to-bottom reading strategy .

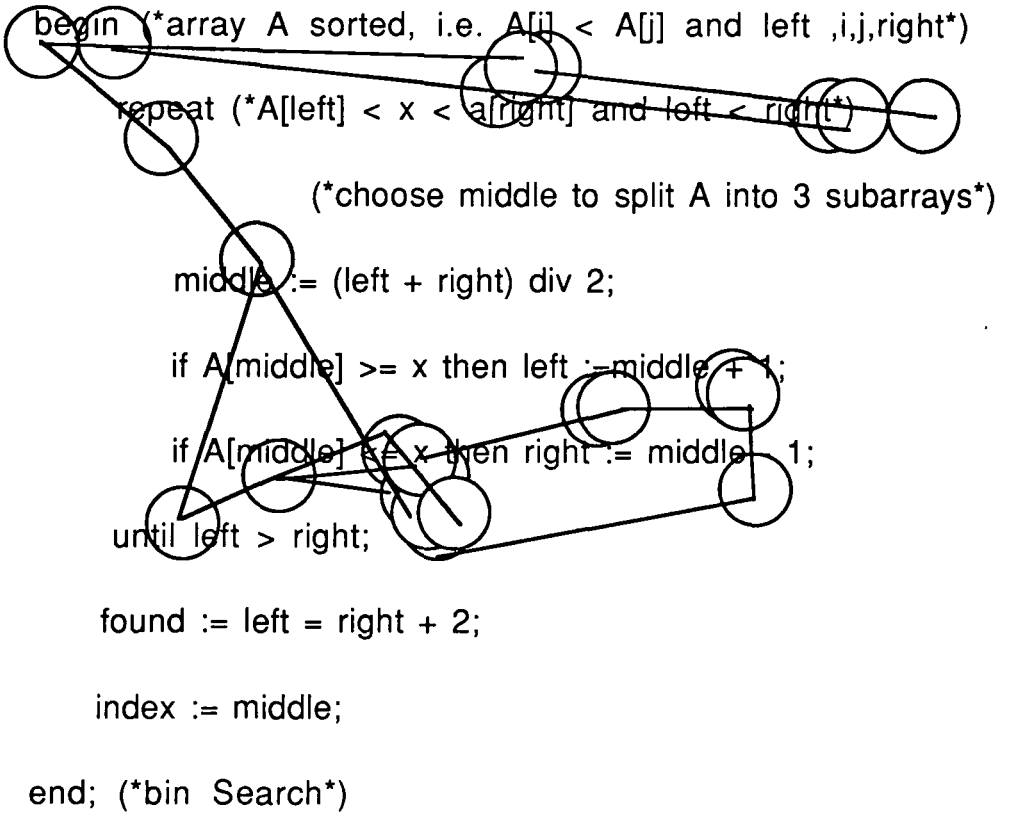


Figure 4.3a. A 'Comparative' reading strategy.

Indeed, the only differences in expertise that were manifest in their study were related to the percentage of time spent viewing particular parts of the algorithm. Crosby and Stelovsky partitioned the algorithm used in their study into a number of areas, which could in turn be grouped into five classes: Comments, comparisons, complex statements, simple assignments and keywords. They then analysed the average percentage of fixation time spent by the novice and expert groups on these five areas. This analysis is shown in figure 4.4. It appears from this analysis that experts spend more time reading complex statements, whereas novices tend to concentrate upon comments. One might infer from this that experts develop their understanding of the program from the program text itself, while novices rely to a greater extent upon comments to augment their understanding. This would in itself seem to be a reasonable assumption. However, it would be

interesting to discover the extent to which experts actually use comments to aid understanding. Moreover, if experts tend not to rely significantly upon comments during comprehension activities then this would clearly question their utility, especially in program maintenance tasks where comprehension clearly plays a key role.

The results of the Crosby and Stelovsky study are rather difficult to account for given the large body of literature which has demonstrated a strong relationship between particular forms of strategy and expertise. In general, their study failed to reveal *any* significant relationships between viewing strategy and programming experience or comprehension. The Crosby and Stelovsky study is interesting since it adopts a rather different methodological perspective to the studies of program comprehension reported above. Moreover, the results of this study are contrary to what one might expect given the results of existing work. In particular, it suggests that reading strategy and viewing time do not display any significant correlation with programming experience. In contrast, the other studies reported in this chapter would predict systematic variations in search strategy associated with differences in expertise. It would appear injudicious to reject previous accounts of program comprehension simply upon the basis of a single study. However, bearing in mind this caveat, it is clear that the Crosby and Stelovsky study raises a challenge to existing theories of program comprehension.

The studies reviewed so far have been concerned primarily with the strategic elements of program comprehension. However, there also exists a considerable body of literature which has addressed issues relating to the strategic aspects of program generation. One problem with knowledge-based theories that was highlighted at the start of this chapter, is that while such theories serve to describe the content of stereotypical knowledge structures, they have typically not attempted to specify the processes that control the transformation of these representations into code during problem-solving. Recently however, Rist (1986a, b; 1989; 1990) has proposed a model of schema creation in programming that describes the way in which plan-based knowledge structures are transformed into programs and accounts for the differences in generation strategy that have been observed to be associated with differences in expertise.

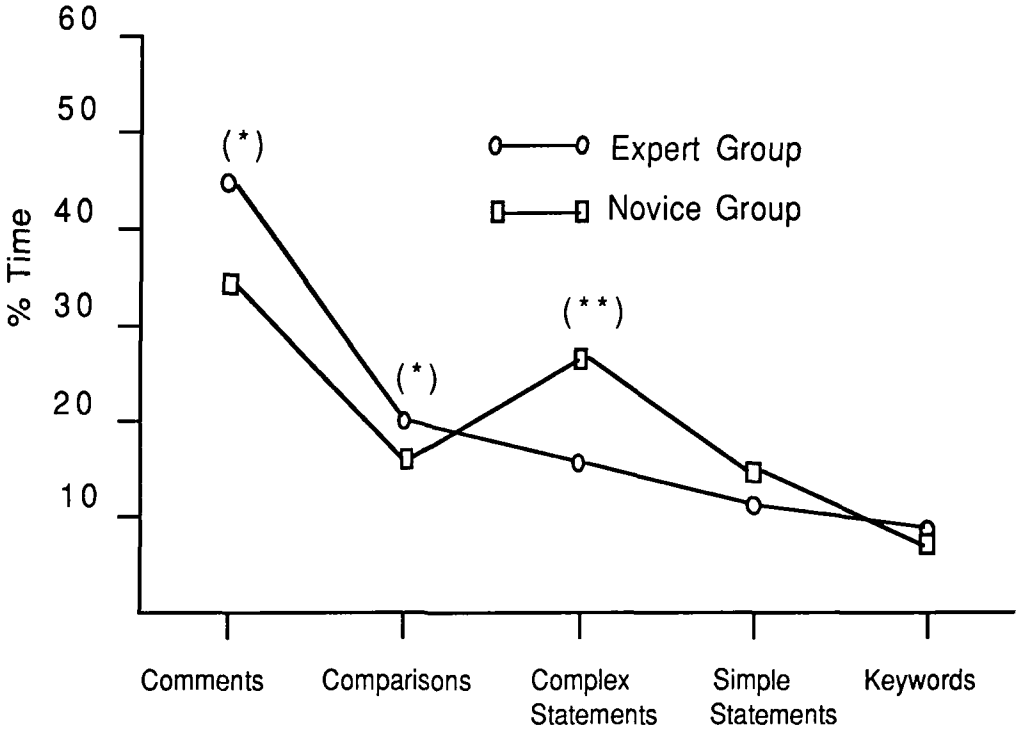


Figure 4.4 . Percentage of fixation time spent by the novice and expert groups viewing the five areas identified by Crosby and Stelovsky.

4.3 Strategies involved in program generation

4.3.1 Rist's model of focal expansion

Rist (1989) has proposed a model of program generation which illustrates how simple fragments of plan knowledge are composed at various levels of abstraction and, in addition, demonstrates how these plan structures are eventually implemented as code-based representations. Unlike previous plan-based accounts of programming behaviour which suggest that plans have a flat structure consisting of a sequence of steps that are simply concatenated to achieve a goal (Detienne and Soloway, 1990; Robertson and Yu, 1990), Rist's model assumes a more complex plan structure and suggests several distinct mechanisms which govern plan composition and use.

The first stage of plan formation, according to Rist, involves creating a single line of code. Rist adopts Mayer's (1987) suggestion that the smallest fragment of knowledge used in program comprehension is a transaction. Transactions conceptualise a program statement in terms of the operations that take place, their location and the object that is acted upon. For example in order to understand the line of code, LET B=A + 1, requires 10 transactions. First the integer and increment must be defined and stored in temporary memory (4 steps), and then added (1 step). Next, the location of the sum must be defined and then deleted from temporary memory (3 steps). Finally, control must be transferred to the next statement and that statement executed (2 steps). According to Rist, the creation of even a single line of code involves a significant amount of reasoning and planning.

Rist proposes that within a single line of code, there will be a central or focal code fragment that represents the most important operation performed by that line . This focus is described as that part of the statement that directly implements the current goal. In the above example the programmer's goal will be to increment a number, hence the code fragment that achieves this (i.e., +1) becomes the focus of the programmers current activity, and other code structures will be built around it.

During the next stage of plan development, single lines of code are combined to create a programming plan. To do this, the plan focus must be extended to include other subsidiary plan elements. Rist suggests that this extension process can take two forms; plan creation or plan retrieval. In plan creation, the plan is developed backwards from the goal, to the focus of the plan and finally to its extension. Here, plan generation begins with the calculation element of the plan and then continues with the initialisation and the output. In plan retrieval, plans are generated in schema order, such that the initialisation part of the plan will be generated first, then the calculation part, and finally its output.

The next level of plan use describes the process of plan composition to form a program. Here, Rist suggests simply that during this stage, basic plans are combined to form more complex plans. Rist claims that the visible structure of a complete program is created by these complex plans which make up the final form of the program. Rist further claims that only at the end of this construction process will the abstract form of the solution be apparent. Here, Rist suggests that this abstract program structure can be analysed in terms of the 'role structure' of various elements of the program. These role structures describe the goals of various program elements in terms of whether these goals involve establishing some input, making a calculation or relate to output in some way.

The model of solution design proposed by Rist describes plan generation at these four levels and makes two basic claims about the effects of knowledge on behaviour. Essentially, if knowledge can be found to guide program design, then top-down and forward plan generation will be observed. If knowledge cannot be found, then a solution will be created by focal expansion, which characterises program generation as a bottom-up process. Hence, the novice, given a goal that they have extracted from the problem statement, will retrieve a fragment of code that directly implements the current goal, and will then construct the rest of a plan around this focal segment. Rist equates the development of expertise with an increase in stored knowledge which specifies the required plans. Given this plan knowledge, Rist claims that experts simply need to retrieve stored schemata and that a programmer's generation strategy will change from focal expansion to schema expansion as expertise develops.

Rist (1989) reports a longitudinal protocol analysis of a small number of subjects (10) who were asked to generate programs to solve a number of problems. Both the verbal statements produced by subjects and their resulting programs were analysed. The coding scheme used to analyse these protocols described, where appropriate, each code fragment or verbal utterance as relating to an "input goal" (I), or to a "calculate goal" (C) or to an "output goal"(O) (Spohrer, Soloway and Pope, 1985). In addition, an analysis of the program's plan structure was undertaken, and the order of emergence of plan elements (expressed as an ICO sequence) was established.

Two sample protocols, illustrating this analysis are shown in figure 4.5. The first protocol illustrates the process of focal expansion, where the focal plan element is retrieved early during during plan development. According to Rist, this form of plan implementation would be observed in the case of novice programmers. Here, the order of generation is contrary to what one would expect if the plan was retrieved and implemented in schema order (i.e., in ICO order). The second protocol shows a program generation episode which reflects forward plan expression in schema order (ICO). This protocol represents the type of coding strategy thought to correspond to expert coding behaviour.

The results of this study provide support for the model of code generation proposed by Rist. Firstly, the protocol analysis demonstrated that both forms of generation strategy, i.e., focal expansion and schema retrieval, were displayed by subjects. Secondly, there was a change in the mode of plan expression with experience. Evidence for this change in plan expression was established by comparing the degree of schema expression in plan creation (i.e., focal expansion) versus plan retrieval. This analysis showed that there was a significant increase in coding and verbal behaviour that could be accounted for in terms of plan retrieval processes associated with programming experience. Correspondingly, there was a significant decrease in plan creation with experience.

VERBAL STATEMENTS	PROGRAM CODE	INTERPRETATION	
(N7) "For rain itself,		goal	O
rain is going to equal ...	+ rain	focus	C
I need ... I need something ...			
I need something there that keeps ...	rainfall:=	extension	C
I'm setting the sum equal to the amount			
of rain from	sum	extension	C
before, OK ... sum will be a problem,			
I'll come back to it later ...			
[other goals solved] ...			
OK, do sum gets sum plus rain, or			
even better	rainsum:=rainsum+rain	focal line	C
... and now set that to zero"	rainsum:=0	extension	I

VERBAL STATEMENTS	PROGRAM CODE	INTERPRETATION	
(N2) "The first thing I want			
to do is to get the original word ...			
You might want to use an array for the		goal	
letters in the word, so you have ...	TYPE letters=array[1,20]		
	of char; VAR word: letters;	define a word	
Now get the original word, so say			
WHILE ... use a REPEAT loop ...			
give a prompt first ...	write ('please enter a word');	extention	I_{read}
then REPEAT ...	repeat	extension	I_{loop}
umm ... you want to read in ...	read(...	focus	$C_{read,loop}$
you want a counter ...			
initialize a counter,	i := 0 [before repeat]	extension	I_{count}
and then you want to repeat	read (word[i]);	use	O_{count}
... so you read it in, and			
then let i ...	i := ...	focus	C_{count}
... intialize i to 1	i := 1 [before repeat]		
... and then increment it by 1	i := i + 1		
... uh-uh, until ..."	until(word[i] = ' ');	use	O_{read}

Figure 4.5 The first Protocol (above) illustrates the process of focal expansion, where the focal plan element is retrieved early during during plan development. This form of expansion is claimed to be characteristic of novices. Here, the order of generation is contrary to what one would expect if the plan was retrieved and implemented in schema order (i.e., in ICO order). The second protocol shows a program generation episode which reflects forward plan expression in schema order (ICO). This protocol represents the type of coding strategy thought to be used by experts.

4.3.2 Studies of change processes - The Parsing/Gnisrap model

Green, Bellamy and Parker (1987a and b) have attempted to extend Rist's model of focal expansion by proposing a comprehensive model of coding behaviour which highlights those features of the device, task, interaction medium and user knowledge that contribute to the use and the development of particular forms of programming strategy. They suggest that there are two important aspects of coding behaviour that a model needs to account for. One aspect of coding behaviour relates to the fact that code is often not generated in a strict linear fashion, where the final text order of the program corresponds to its generative order. For instance, Figure 4.6 shows a number of departures from linear generation for a typical Basic protocol. Green et al suggest that these departures from linear generation indicate problems in the task and the device languages or their relationship. They suggest that departures from linear generation are significant, since they increase mental workload, may give rise to omissions and oversights and, depending upon the editor used to create the program, can involve many additional keystrokes to effect successful navigation around the code.

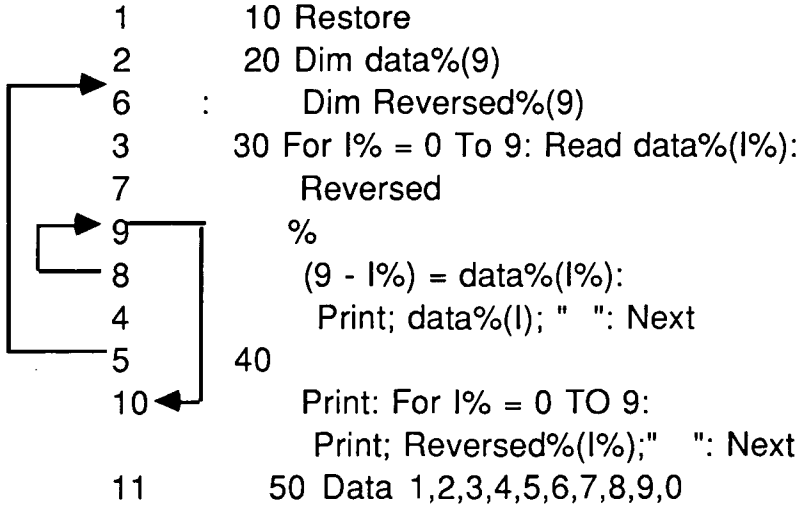


Figure 4.6. Departures from linear generation for a typical Basic protocol. The numbers on the left represent the order of generation.

A second aspect of coding behaviour relates to the ease with which information can be extracted from the text structure of the program. When programmers adopt a non-linear style of code generation they need to be able to read and to comprehend earlier parts of the code before they can insert subsequent parts. This re-comprehension process is likely to be facilitated if the programmer can easily extract relevant information from the program text.

Green et al introduce the 'Parsing-Gnisrap' cycle to describe the cyclical alternation between code generation and recomprehension. The parsing-gnisrap model describes two fundamental psychological processes which underpin coding behaviour. Firstly, programming text is mentally elaborated from a skeletal plan, and when working memory is full, or when overload is threatened, part of this text is output to an external medium. When the programmer subsequently needs to recover the details of parts of the text that have been externalised, the text must be comprehended. To achieve this the text must be parsed in order to recreate the original plan structure. Gnisrap is simply the reverse of parsing and involves transforming an internal representation into an externally represented text structure.

This model borrows directly from that proposed by Rist in that it suggests that expert coding is a forward chaining process which involves matching elements of a skeletal plan to programming plans and then expressing these plan elements in code-based terms. Rist's model suggests that expert programmers will tend to generate code in plan order, where the order of generation will reflect the final text order of the program. Conversely, the parsing-gnisrap model makes rather different predictions about the order of program generation. These differences arise as a consequence of one primary extension to Rist's model.

The behaviour of the parsing-gnisrap model, unlike Rist's model of focal expansion, is fundamentally determined by working memory limitations. In Rist's model, programming knowledge is represented as schemata-based structures which state the code, its purpose, the role of components within the code, and its pre and post conditions. Plans are built by retrieving a focal line and then by subsequently extending this focal structure to include the plan's subsidiary code. In addition, other schemata may be invoked at this time by matching the

preconditions of the present schemata with the pre and post conditions of other schemata. According to Rist's model, nonlinearities will only occur when a programmer has to interleave a new plan into the existing program. The parsing-gnisrap model allows greater flexibility by suggesting that nonlinearities can occur within the generation of individual plans. For instance, it suggests that the minor parts of plans will often be omitted during generation, and that programmers will need to return to complete code fragments after generating later parts of the program.

The parsing-gnisrap model suggests that rather than build up a whole program internally and then output it to an external source, programmers will output fragments as they are completed, or will dump incomplete fragments when working memory becomes overloaded. The model deletes information from working memory when it is output to an external source. Hence, when the programmer subsequently needs to refer to an existing code fragment, it will need to be parsed such that its original plan structure can be recreated.

The behaviour of the parsing-gnisrap model (and correspondingly the behaviour of the programmer being modelled) depends upon characteristics of the device, the task language, the interaction medium and the model's knowledge representation. For example, the device language might represent the commands necessary to manipulate editor functions. If one has to navigate a complex structure in order to insert or to comprehend information, as necessitated by the parsing-gnisrap model, then the ease or difficultly of doing this is likely to affect strategy.

The use of a particular interaction medium e.g. pen and paper, VDU, will also affect strategy. Green et al illustrate this by comparing the ease of using pen and paper to generate a program as opposed to dictating a program. With pen and paper it is a simple matter to insert new material, for instance, to balance parentheses or declare new variables. According to Green et al, pen and paper has a large 'access window', which allows the programmer to access any part of the expression being built. Conversely, in dictation, the access window is restricted to a single point. Between these two limiting cases lie word processors and text editors where the user must do some work to access any point other than the current cursor location.

The size of the access window is a property of the interaction medium, not of the task language. Hence, one can dictate or write Pascal or algebra and here the interaction medium differs while the task language remains the same. However, Green et al claim that the importance of the window size depends upon the task language. When using languages with many left-to-right constraints such as algebra or Pascal, it is important to have a wide access window, whereas other notations such as reverse Polish, which have fewer constraints of this sort, may not demand a large access window.

Characteristics of the task language will also affect strategy. For instance, some languages are highly resistant to local change. Such languages might be described as viscous (Green, 1990a and b; 1991). For example, it is harder to insert a new section into a Pascal program than into an equivalent Basic program. This phenomenon is evident in the results of an experiment carried out by Green et al to test certain predictions stemming from the parsing-gnisrap model. This experiment is described in more detail in chapter 5, however its main finding was that the extent of non-linear generation in coding was related to the language being used. Hence, Pascal programmers adopted a highly non-linear style of code generation whereas both Basic and Prolog programmers departed infrequently from a linear generation strategy.

Green et al claim that there may are several possible reasons for these difference in strategy. Firstly, Pascal procedures may offer better facilities for a non-linear approach than Basic subroutines. Secondly, the Pascal teaching tradition, and its emphasis upon stepwise refinement, has always emphasised a top-down, and consequently non-linear method of program development. Thirdly, Pascal appears to be inherently viscous, and an approach which minimises interleaving will be preferred. Finally, Basic is less role-expressive than Pascal (see chapter 5). Hence, Basic programmers will experience more difficulty comprehending code once it is generated and will therefore adopt a strategy which minimises the amount of re-comprehension that is necessary, i.e, a linear generation strategy.

The factors affecting coding strategy which are outlined above are, with the possible exception of Pascal teaching tradition, related explicitly to the formal structure of the task language. However, it is clear that strategy will also be

determined to some extent by the structure of the user's knowledge of how to perform programming tasks. In plan-based theories of programming expertise, plan structures are uniform and all their parts are equally accessible. However, it is clear that, in most cases, these plans will be composed of a major part and other necessary, yet minor elements. For example, a summation plan will require a minor initialisation component and more salient summation element.

Green et al suggest that "These minor and possibly less salient parts of plans are sometimes left out by programmers during the first pass, either intentionally or by mistake, and are inserted later. On the other hand, it is extremely rare to see the minor part included in the first pass and the major part omitted. This asymmetry suggests to us that the effect is caused by asymmetry in the knowledge structure: the major part is focal to the plan, but the minor part is invoked only as a precondition required by the major part" (p 138).

The importance of the parsing-gnisrap model is that it describes the process by which a plan is instantiated in a programming notation and the effect that various notational features will have on plan implementation and general coding strategy. In this sense, the parsing-gnisrap model complements and extends Rist's work on focal expansion by detailing the mechanisms that control plan use to give rise to various forms of strategy. For instance, the Green et al work showed that Pascal programmers frequently made backward jumps during coding to insert base preconditions, whereas in Basic, this retrospective inclusion of preconditions was infrequently observed. Similarly, PROLOG programmers almost always inserted a base case before developing other parts of the associated procedure. Green (1990b) suggests that this may arise because, in PROLOG, base cases are spatially contiguous with their associated focal line in the main case, whereas in Pascal, plan structures are diffuse.

While the parsing-gnisrap model displays some overall similarity to the model of focal expansion proposed by Rist, it clearly makes different predictions about the order of plan generation. In particular, Rist's model would predict that experts would generate code in schema order, while novices would develop code by focal expansion. The parsing-gnisrap model makes no predictions about the relationship between expertise and generation strategy. However, the main factor that gives

rise to nonlinearities in this model relates to its limited working memory capacity. Since there is no reason to expect that experts would have a greater working memory capacity than novices, then, according to the parsing-gnisrap model, they should display broadly similar generation strategies, and hence exhibit similar patterns of nonlinearities.

Moreover, the data used to support the parsing-gnisrap model was collected from expert programmers. As we have seen, this data appears to show that in many cases code is not generated in plan order, as would be predicted by Rist's model. For instance, in Pascal, preconditions are often inserted after other parts of a plan have been developed. Since Green et al did not analyse the plan structures evident in the programs generated by their subjects, it is not clear that one can make a direct comparison between this data and that reported by Rist. However, in general terms, the nonlinearities observed in the Green et al study do not appear to support a model of expert coding behaviour such as Rist's which would predict that plans will be retrieved and implemented in the order in which they appear in the final program.

4.3.3 Knowledge restructuring: Extensions to the parsing-gnisrap model and the role of focal expansion

In chapter 8 an experiment is reported that attempts to extend the paradigm suggested by Green et al by examining the nonlinearities in program generation made by programmers of varying skill level using different languages. This experiment suggests that the level of skill of the programmer is a more important discriminator than the programming language used. However, there were clear interactions in this experiment between skill level and language. This result is interpreted within the broad framework suggested by this thesis which advances a model of knowledge restructuring to account for skill development in programming.

In particular, it is suggested that certain languages may facilitate the implementation and comprehension of focal code structures. This would explain the interaction between expertise and language. Hence, we might assume that as

knowledge representation is in the process of development, any additional means of facilitating a preferred programming strategy, such as might be provided by features of the notation of a particular language, would be of importance. At higher levels of skill, factors relating to the organisation of knowledge appear to play a greater role in the determination and support of particular forms of strategy. This experiment might be seen as providing a bridge between Rist's work on knowledge representation and the emphasis of the parsing-gnisrap model on language features, by suggesting possible ways in which these factors might interact to determine the nature of programming strategy.

Additional support for this model of knowledge restructuring is provided by two further experiments which are reported in chapters nine and ten. In chapter nine, an experiment is described which examines the temporal generation of code for novice and expert programmers. Here, following Rist, each statement of the completed program was classified according to whether it constituted a focal or a non-focal line. The model of knowledge restructuring presented in this thesis would suggest that a programmer's representation of programming knowledge is not uniform, as is suggested by plan-based accounts. Rather, certain important code structures, i.e., focal lines, will be represented with greater saliency than other subsidiary plan elements. Moreover, in common with the parsing-gnisrap model, it is suggested that focal lines will commonly be generated first during coding (i.e., before non-focal lines). This reduces the load on working memory and provides a framework around which the rest of the program can be built.

Since this knowledge restructuring process is assumed to underpin the development of expertise in programming, one would expect to see differences in the temporal generation of focal vs non-focal structures corresponding to the level of expertise of the programmer. This prediction is supported by the results of the experiment reported in chapter nine. In particular, expert programmers were seen to generate significantly more focal lines during the early stages of the development of a program, whereas novice programmers generated significantly more non-focal lines. This disparity was maintained until quite late in the development of a program, but focal and non-focal line generation tended to converge towards the final stages of program generation. This provides some support for the idea that expertise is related to the restructuring of programming

knowledge, since one would expect the saliency of various plan components to affect the generation patterns evident during coding.

In addition, it is suggested that the early generation of focal lines during coding reflects the adoption of a top-down hierarchically levelled approach to program development. In chapter 6, an experiment is reported which suggests a relationship between design experience and subjects' ability to use cues to extract plan structure from a program text. It may also be the case that design experience can facilitate particular generation strategies. In particular, focal lines may represent a discrete level of design abstraction.

This would certainly accord with Rist's characterisation of focal lines. According to Rist (1989) "the (plan) focus...marks the start of detailed design in the domain of the program" (p. 403). Most design methodologies embody an approach which suggests that programs should be developed in a top-down fashion, beginning with the highest level of design abstraction and progressing to lower levels only when the preceding levels are fully articulated. Hence, programmers with design experience may be inclined adopt a strategy which causes them to articulate focal lines during the early stages of program development.

Further evidence for the knowledge restructuring argument voiced by this thesis is reported in chapter ten. Here, an experiment is reviewed which employs a program memorisation and probe recognition task to explore the form of knowledge representation for programmers of various skill levels. In this experiment, subjects were presented with a number of small Pascal programs which they were asked to memorise. Subsequently, subjects were presented with a probe item and were asked to state whether it was contained in the original program. Half of these probe items were derived from the original programs and the other half from similar programs. Additionally, half of the probe items derived from the original programs were classified as focal lines and the other half as non-focal lines. The results of this study showed that intermediates and experts exhibited approximately the same recognition accuracy, suggesting that they were able to access similar knowledge structures. However, the expert group correctly recognised focal lines much faster than the intermediate group. This is taken as evidence for the idea that experts represent focal lines with greater saliency than

their less experienced counterparts and, consequently, as evidence for the model of knowledge restructuring presented in this thesis.

These experiments suggest a possible means of linking the process of focal expansion described by Rist with the idea that code is generated and evaluated in cyclical phases as suggested by the parsing-gnisrap model. However, the knowledge restructuring model presented here differs from previous work in a number of respects. Firstly, in contrast to Rist's model, focal expansion is seen as a strategy which arises due to knowledge restructuring, and hence will be exhibited predominantly be experts rather than by novices. Secondly, the work reported in this thesis predicts that differences in generation strategy will be related to differences in expertise. In contrast, while the parsing-gnisrap model makes no specific predictions about the effects of expertise, this model implies that there will be no relationship between these two factors. Generation strategy, according to the parsing-gnisrap model, depends primarily upon working memory constraints and upon the nature of the task language being used, and there is no reason to believe that these would, in turn, be related specifically to expertise.

4.3.4 Change-Episodes

The cyclical nature of code generation and comprehension suggested by the parsing-gnisrap model is also reflected in other studies of programming behaviour. For instance, Gray and Anderson (1987) have carried out an analysis of so called 'change episodes'. Change episodes are described as key junctures in the coding process where programmers alter their code. Gray and Anderson suggest that an analysis of these episodes can help to illuminate the cognitive process involved in programming, since they provide a wealth of information about the programmer's goals, about their planning activities, about their evaluation of existing code and about the error detection and correction mechanisms which are typically invoked during programming tasks.

Gray and Anderson suggest that coding should be viewed as a problem solving process where the problem statement represents the initial state of the problem and the completed program its goal state. The knowledge and skills of the programmer

determine which parts of the problem involve problem solving and which parts simply entail the retrieval of information from long term memory.

Following Allwood (1984), Gray and Anderson distinguish two types of problem solving activities: progressive and evaluative. Progressive activities work directly toward the goal state of the problem, whereas evaluative activities involve checking some already executed part of the problem solution. In programming terms, coding (and planning what to code) are progressive activities and checking and changing code are evaluative activities - in Gray and Anderson's terminology, change-episodes.

Previous accounts of problem solving behaviour have, according to Gray and Anderson, tended to ignore these evaluative activities and studies of programming appear to have followed this trend. For example, most accounts of debugging (Vessey, 1985; 1986; 1989; Waddington and Henry, 1989) involve progressive activities. The issue of how programmers actually evaluate their progress during debugging and coding is not normally addressed. The work of Green et al does suggest an important role for evaluative activities during coding, however the parsing-gnisrap model does not make any strong predictions about when and where these activities will take place. By relating change-episodes to predicted planning or problem solving difficulties during coding, Gray and Anderson have been able to specify the circumstances in which change-episodes are most likely to occur, and this provides a systematic basis for understanding the evaluative activities that normally take place during coding.

The Gray and Anderson study involved analysing the change-episodes contained in protocols produced by 'advanced novice' subjects writing a single LISP function. A change-episode is said to occur when the programmer alters some code that they have already written or when they subsequently modify a plan that has already been articulated (derived from a verbal protocol). Gray and Anderson suggest a tripartite analysis of change-episodes involving the change goal, the noticing event and the fix. The change goal is an articulated goal that the programmer later decides to change. The noticing event describes the first indication that the programmer wishes to change the goal structure of their solution. The fix is the goal structure after it has been changed.

Gray and Anderson derive a number hypotheses which relate to each of the three elements in their change-episode analysis. As far as change goals are concerned, Gray and Anderson suggest that the probability of a goal being the target of a change-episode will be correlated with the amount of planning involved in its execution. This hypothesis is based upon the assertion that goals which involve a significant amount of planning arise because the goal is not yet well learned and has not been subject to knowledge compilation or that there are so many variations as to how the goal can be used that a specific rule for the required variation has not yet been compiled. They suggest that in the first case, poorly learned goals can easily be coded incorrectly and hence they can become the target of evaluative activities. In the second case, choosing the correct instantiation of a goal will depend upon having a clear idea about how the rest of the function will be coded, and this in turn can depend upon a number of highly variable goals.

The second element of Gray and Anderson's change-episode analysis, noticing events, are indicated either by keystroke analysis or from verbal protocols. In the first case, the programmer might interrupt their current activity in order to backtrack to an earlier part of the program. In the second case, verbal utterances might be indicative of a noticing event (for instance, an "oops" followed by a revision) or it might consist of a full articulation of the change-goal.

Gray and Anderson suggest three issues relating to the nature of these noticing events which form the basis of their hypotheses. Firstly, do change-episodes form an abrupt interruption to the progressive activity of coding or do they arise as a more natural outcome of an activity such as symbolic execution that was begun for some other reason? Secondly, do the nature of change episodes differ if they are initiated by different types of noticing events? Finally, is there a relationship between the activity that immediately precedes the noticing event and the change goal?

The final part of Gray and Anderson's analysis is related to *how* a goal is altered during a change-episode. Essentially, the fix is the goal-structure after it has been changed, and this is established by comparing the goal structure before the change-episode with the goal-structure after the change-episode. They suggests

two issues that relate to fixes. Firstly, are their as many types of changes or fixes as there are change-episodes, or can most fixes be categorised? Secondly, what does the nature of these fixes suggest about the role of planning in coding?

The results of the Gray and Anderson study support the specific hypotheses that are outlined above and, in addition, provide a basis for a discussion of the more general questions raised by their analysis. For example, they found that the probability of a goal being the target of a change-episode was indeed correlated with the amount of planning involved in its execution. Their results also suggest that only a small number of fix categories exist. In particular, their results indicate that a large proportion of fixed change-episodes could be grouped into just two fix categories; one which involved relatively minor editing of a goal's subgoals and one which entailed major transformations in the goal's structure. Their results also suggest that a change-episode can be initiated in three distinct circumstances; as an interrupt to coding, as tag-along to other change-episodes or as a product of symbolic execution.

The first of these describes the situation where programmers simply stop coding to make changes. Gray and Anderson suggest that these interrupts appear to be part of the planning process. They argue that programmers begin to work on a goal before it is fully planned and that this may help programmers to remember the goal structure of the plan without committing them to its detailed elaboration. It appears that as programmers begin to elaborate a plan, this may suggest that higher order goals require revision.

This process clearly has similarities with the kinds of activities described by the parsing-gnisrap model, since it relates planning and working memory limitations to the fragmentary nature of code generation and accounts for the evaluative activities that need to be undertaken to recover or to recreate higher-level goals.

Change-episodes can also be initiated as a result of tag-alongs to other change-episodes. In general, these episodes involve fairly straightforward fixes such as correcting spelling mistakes or balancing parentheses. However, some episodes appeared to arise due to confusion over which of several methods to apply in a particular situation. Here, Gray and Anderson argue that such methods

may at an intermediate stage of compilation, where the programmer may know several methods, but not know which to apply. Finally, change-episodes can occur as a result of symbolic execution where the programmer had been symbolically executing their code immediately prior to noticing the need to change a previously coded or stated goal.

The evidence for these three categories of change-episode was mainly derived from verbal protocols. However, Gray and Anderson also present collateral information (i.e., time between last keystroke and noticing event) which provides additional independent support for their categorisations. In the three categories they found that the mean time in seconds between the last keystroke and the noticing event was 52.5 (symbolic execution), 2.2 (interrupts), and 4.0 (tag-alongs). These differences are statistically significant, and provide additional support for the validity of their change-episode categorisation.

Gray and Anderson's change-episode analysis has proved important since it makes firm theoretical predictions about where change-episodes will occur during coding. In addition, it provides a detailed account of the cognitive mechanisms which underpin such change processes and other more general evaluation activities. Moreover, the change-episode analysis presented by Gray and Anderson clearly ties in with other work on change processes, and in particular with the parsing-gnisrap model. Both of these models suggest that because of certain cognitive limitations, code is generated in fragments and that these fragmentary structures need to be evaluated in some way in order to recreate the original plan or goal structures that initially gave rise to the generation of these code fragments.

However, despite the close parallels between these models, both display a very different emphasis. The parsing-gnisrap model is primarily concerned with the effects of features of the task language on coding behaviour, and has little to say about knowledge representation. Conversely, the change-episode analysis attempts to relate the evaluative activities that occur during coding to a specific model of the planning and problem solving processes that underpin programming. Viewed together, these models clearly advance our understanding of problem solving in programming by illustrating the various factors that give rise to the

adoption of particular forms of strategy. This analysis leads to a characterisation of programming behaviour which highlights its true complexity and distinguishes it from other domains.

Research on change processes such as that used to support the parsing-gnisrap model and Anderson and Gray's analysis of change-episodes is currently lagging some way behind advances in language design. For instance, the object oriented programming paradigm has spawned a number of languages which have been designed specifically to support an information structure which facilitates code change and modification. The emphasis of these languages is upon code re-use, and to make re-use possible these languages need to exhibit sufficient flexibility to allow the programmer to easily modify and tailor the code. However, there is little empirical evidence which demonstrates the extent to which particular object oriented languages might support change and re-use (See, Green, Gilmore, Winder and Davies, 1992).

Lange and Moher (1989) report a single subject study of a professional software developer working in an object oriented environment which showed that code was frequently re-used. However, their subject achieved this by simply copying text rather than using the method-inheriteance mechanisms provided by the system. Lange and Moher suggest that this "reflects an overall approach of comprehension avoidance, in spite of the fact that the subject was modifying code that she herself had written earlier" (p. 69). Similar findings have arisen in other studies of software reuse (Maiden and Sutcliffe, 1991; Sutcliffe and Maiden, 1990). If these findings are generalisable, then they may suggest that language designers will need to think again about supporting change processes in object oriented environments. Moreover, this finding provides additional support for the idea that change processes are cognitively significant and that such processes require some cognitive effort to effect.

4.3.5 Characterising coding activities and program design as opportunistic

The studies of change processes that are reviewed above appear to suggest that coding may be considered to be an opportunistic process (Hayes-Roth and

Hayes-Roth, 1979; Hayes-Roth, Hayes-Roth, Rosenschein and Cammarata, 1979) consisting of bouts of activity each of which involve the creation of code fragments. These fragments are, in turn, continually reevaluated and modified in respect to the particular goal or subgoal currently under consideration (Green et al, 1987). In addition, the development of code may be postponed at any time in order that the programmer might direct her attention to other goals or subgoals, possibly in response to the recognition of previously unforeseen interactions between code structures (Gray and Anderson, 1987).

Such opportunistic strategies can be contrasted with top-down models of problem solving which suggest that problem solving is a focused process that starts from high level goals which are successively refined into achievable actions. In addition, it is claimed that this process is hierarchically levelled. That is, plans or sub-goals are always fully expanded or refined at the same level of abstraction before the problem solver or planner moves on to a lower level in the plan/goal hierarchy. Such a view of problem solving is well established in the psychological literature (Kant and Newell, 1984; Newell and Simon, 1972) and hierarchical planning models in artificial intelligence are founded upon these principles (Ernst and Newell, 1969; Sacerdoti, 1974; 1977).

More recently an alternative view of the planning/problem solving process has emerged. This view characterises planning and problem solving as opportunistically mediated, heterarchical processes (Hayes-Roth and Hayes-Roth, 1979; Hayes-Roth et al, 1979). Here, in contrast to top-down models, planning is seen as a process where interim decisions in the planning space can lead to subsequent decisions at either higher or lower levels of abstraction in the plan hierarchy. At each point during the planning process the planner's current decisions and observations may suggest various opportunities for plan development. For instance, a decision about how to conduct an initially planned activity may highlight constraints on later activities, causing the planner to refocus attention on that part of the plan. In a similar way, low-level refinements to an abstract plan may suggest the need to replace or modify that plan. Hayes-Roth and Hayes-Roth (1979) have provided support for their view of the planning process by observing subjects planning a series of errands through a town. Here subjects tended to mix high and low-level decision making. Often subjects planned

low-level sequences of errands in the absence or in direct violation of a high-level plan.

The dichotomy between top-down and opportunistic processing is also evident in empirical studies of the programming activity. For instance, Jeffries et al (1981) found that both novice and expert programmers decomposed their designs in a top-down fashion - moving between progressive levels of detail until a particular part of the solution could be directly implemented in code. One major difference between novice and expert programmers was that novices tended to employ a depth-first search of the solution space - expanding only one part of the solution at progressive levels of detail - while experts adopted a breadth first approach - synchronously developing many sub-goals at the same level of abstraction before moving to a lower level.

Adelson and Soloway (1985) provide additional support for the use of top-down design by experts working in both familiar and unfamiliar domains and Anderson and others (Anderson, Farrell and Sauers, 1984; Pirolli, 1986; Pirolli and Anderson, 1985) have shown that novices characteristically develop designs in a top-down and depth-first manner. Such a view of the design process is also clearly implicated in prescriptive accounts of the software design activity. For instance, the emphasis on top-down problem decomposition and stepwise refinement that is advocated by the structured programming school (Dahl, Dykstra and Hoare, 1972; Wirth, 1971).

In contrast, a number of more recent, studies have highlighted the opportunistic nature of program design tasks. For example, Guindon (1988;1989) found that software designers often deviate from a top-down, stepwise refinement strategy and tend to mingle high and low-level decisions during a design session. Hence, designers may move from a high level of abstraction - for instance, making decisions about control structure (e.g., central vs distributed) - to lower levels of abstraction, perhaps dealing with implementation issues. Guindon notes that the jumps between these different abstraction levels do not occur in a systematic fashion, as one might expect from hierarchically levelled models, but instead can occur at any point during the evolution of a design.

Other studies (Ratcliff and Siddiqi, 1985; Siddiqi, 1985, Siddiqi and Ratcliff, 1989) suggest similar deviations from a simple top-down model within program design. In addition, these studies have questioned the basic adequacy of prescriptions stemming from the structured programming school, in particular the notion of functional decomposition/stepwise refinement. For example, Siddiqi and Ratcliff, (1989) looked at the problem decomposition strategies employed by programmers during program design tasks. They found that subjects trained in structured programming do not carry out problem decomposition in a manner which reflects a search for appropriate levels of abstraction in a specification, as would be expected if these subjects were rigorously applying structured programming techniques. Rather, their problem decomposition is guided largely by various availability effects (Tversky and Kahneman, 1973) derived from the problem representation.

They found that problem decomposition can be affected by stimulus availability or by knowledge availability. These describe cues derived from two sources; Stimulus activated; i.e., where decomposition is motivated by content and surface characteristics of the problem specification, or knowledge activated; i.e., where decomposition is triggered by design experience. The last of these availability effects would suggest a hierarchical problem decomposition strategy, if the subject had received prior training in program design. However, the stimulus activated availability effect may suggest other decomposition strategies, and Siddiqi and Ratcliff have observed that on many occasions problem decomposition is triggered in this way, and can often lead to simplistic problem decompositions. Moreover, they found that this bias towards simplistic decompositions can be reduced by instituting small changes to the problem specification.

The Siddiqi and Ratcliff studies suggest significant problems with hierarchically levelled, top-down characterisations of coding behaviour and software design such as those prescribed by the structured programming school. In addition, if one considers this work in the light of other studies which have attempted to characterise design and programming behaviour as opportunistic, then this clearly also poses problems in terms of applying classical models of problem solving to programming activities. Such models clearly advance a top-down description of problem solving and embody hierarchically levelled goal decomposition methods.

However, many programming activities appear to display few of the characteristics that have being posited by these models. For instance, programs are often constructed in a piecemeal fashion with frequent redesign and evaluation episodes and programmers are frequently observed to move between different hierarchical levels at various arbitrary points during the coding process.

4.4 Studies of Novices

The emphasis of the work that has been reviewed so far in this chapter has been concerned with the nature of the strategies employed by experts - often professional programmers - in program comprehension and generation. These studies suggest that there is an important strategic element in programming skill. We can contrast this view of expertise with knowledge-based theories which suggest that the development of skill in programming simply involves acquiring and building a body of programming knowledge. One of the main implications of knowledge-based theories is that one way of teaching programming involves passing on the knowledge used by experts to beginning programmers, and it follows from this that novice difficulties in programming arise simply from a lack of programming knowledge. However, a number of recent studies suggest that novice programmers display strategic as opposed to knowledge-based difficulties.

For instance, Perkins and Martin (1986) conducted a series of interviews with novice Basic programmers in order to explore the kinds of difficulties they experienced. They suggest that one way of exploring novice difficulties is to ask what students typically know and do not know. In this context, they draw a distinction between low-level programming knowledge - that is, knowledge of particular language structures or stereotypical plans - and higher-level strategic knowledge of the abstract and general tactics of problem-solving. Perkins and Martin suggest that these sources of knowledge form two extremes on a continuum, and they ask whether novice difficulties arise from a beginners shortfall in low-level knowledge structures or from deficiencies in their high-level strategic repertoire.

They suggest that there are two extreme cases where these deficiencies might be manifest. It may be the case that novice programmers have the general cognitive skills for the tasks they face, but that their mastery of the primitives of the language may be poor. Hence, such skills cannot be applied effectively to solve programming problems. Conversely, novice programmers may posses knowledge of the language but be unable to muster this knowledge since they lack the necessary tactical skills. This, of course, is an over-simplification of the problems that novices might face. The novice may know some things about the language and not know other things. Perkins and Martin suggest that common experience testifies that often a person does not simply "know" or "not know" something. Rather, people can often only recruit fragmentary or fragile knowledge.

The term fragile knowledge is introduced by Perkins and Martin to describe knowledge which a student has, but fails to use when it is needed or when it is appropriate. From their analysis of interviews with novice programmers, Perkins and Martin suggest that fragile knowledge may take a number of forms. For instance, missing knowledge is knowledge that the student has either not retained or has never learnt. Inert knowledge is knowledge that the student has but fails to retrieve when it is required. Misplaced knowledge denotes circumstances in which a student uses knowledge inappropriately and conglomerated knowledge signifies situations where a student combines disparate elements of knowledge in syntactically or semantically anomalous ways.

Fragile knowledge does not simply arise because a student has not been taught about a particular programming construct, and evidence for the existence of fragile knowledge is based upon the fact that on nearly 50% of occasions where hints were given to students while they writing programs, the student went on to successfully solve the problem, even though the hint did not make reference to the appropriate knowledge. Their interview data also suggests that these categories of fragile knowledge are "exacerbated by a shortfall in elementary problem-solving strategies" (p 225).

For instance, one example of a 'neglected strategy' is a 'close-tracking' or 'parsing' strategy which involves reading the code in order to discover what the program actually does. One might expect even novices to employ such a

rudimentary problem-solving strategy. However, it appears from the interviews conducted by Perkins and Martin, that novices rarely engage in close-tracking - that is, they infrequently read what exists of a program in order to check their progress in relation to stated or implicit objectives. Perkins and Martin go on to suggest that the use of such elementary problem-solving strategies should be encouraged. They claim that students would gain much by the self-prompting of a close-tracking strategy. They suggest that students should be encouraged, through their programming education, to ask questions like "what will the code I have just written really do?" or "how did my program get that wrong answer?"

The strategic difficulties encountered by novices in the Perkins and Martin study suggests that the acquisition of strategic skills may play a significant role in the development of expertise. It may indeed be the case that novices know appropriate plans but not how to use them. The proponents of knowledge-based theories of programming expertise have generally ignored the strategic aspects of programming skill. However, it is clear from the studies reported in this chapter that strategic elements of programming skill may, in some cases, by of greater significance than its knowledge-based components.

For example, Gilmore (1990) reports an unpublished study which also suggests that novices may know plans but not be able to use them successfully. Gilmore gave novice POP11 programmers a programming problem in which they could choose between an iterative or a recursive solution. The novices in this study had been taught about both recursion and iteration. In a number of cases, Gilmore observed that the student's response to an error message was simply to switch from a recursive plan to an iterative plan or vice versa.

He reports one particular protocol where a subject developed an iterative solution, but omitted the necessary initialisation. This turned out to be the only error in this subjects solution, but rather than try to edit the program, the subject simply switched to a recursive solution. In developing this recursive solution, the subject made an analogous error, by failing to implement a function that returned a value from the stopping condition. The subject then reverted back to an iterative plan. In total, this subject switched between these two possible solutions five times before detecting the mistake. Thus, while this subject obviously displayed some

knowledge of recursion and iteration, it seems that he was unsure about how to use this knowledge in an appropriate fashion.

4.5 Conclusions

The work reviewed in this chapter illustrates the wide range of perspectives that have been adopted by studies which have addressed the strategic aspects of programming skill. Unfortunately, no clear picture emerges from this work. While many studies have clearly demonstrated the importance of strategic knowledge, they have not presented a unified characterisation of the nature of programming strategy. For instance, a number of studies of program comprehension suggest that programmers search through the text of a program in a selective fashion, attempting to pick out salient structures or beacons which can be used to futher guide search or to provide evidence for specific hypotheses about program function which are generated during the comprehension process. While many resarchers would adhere to this general position, there is some evidence that search strategies vary considerably between individuals and that particular forms of strategy do not appear to be realted to either comprehension success or to differences in expertise.

Similarly, models of generation strategy have described programming behaviour in different ways. In addition, one can derive different, and often conflicting, predictions from these models about the strategy that will be adopted by programmers. For example, Rist's model of program generation makes very specific and detailed predictions about the relationship between the order of program generation and expertise. Conversely, the models proposed by Green et al. and by Gray and Anderson, while not explicitly concerned with factors relating to programming skill, suggest that there will not, in general, be a relationship between generation strategy and expertise.

The work reported in this thesis attempts to provide a conceptual bridge between a number of the studies reviewed in this chapter. In particular, the model of knowledge restructuring that is proposed in the final cahpter of this thesis adopts the general framework suggested by the parsing-gnisrap model in order to account

for the cyclical nature of code generation and evaluation that has been observed in a number of the experimental studies that are reported later in this thesis. In addition, this restructuring model relies centrally upon Rist's elaboration of the process of focal expansion in order to explain how representations of programming knowledge can be transformed and eventually implemented as programs. The intention of this work is twofold. Firstly, to provide a model that can account parsimoniously for a wide range of experimental findings, and secondly to provide a theoretical basis which might allow some integration of previously disparate areas of research. Hence, this model, and the experiments reported in this thesis, attempt to account for particular forms of programming strategy by demonstrating how strategy might be determined by the programmer's knowledge representation, by features of the task language and via their interactions.

Notes

1. Software metrics (or more accurately program complexity metrics, since other metrics measure different things) are intended to provide a measure of the complexity of program code. Such metrics are normally derived from a simple count of a number of program surface features, i.e, number of operators or operands, etc.

2. One can again draw parallels here with work in text comprehension, particularly Rumelhart's (1977) work on story summarisation.

Chapter 5. The role of task language/notational features in programming strategy and program comprehension

5.1 Introduction

In chapter 2 we suggested that to fully understand problem solving in complex interactive domains such as programming we need to consider the effects of task and device language features on problem solving behaviour. In this context, Payne (1987) suggests that problem solvers need to maintain two problem spaces, a goal space and a device space, and that they need to construct a mapping between the two. This approach to understanding problem solving in complex environments differs from more traditional accounts of problem solving behaviour in two ways. On the one hand, operators are not applied directly to objects, rather they are effected via a task language which maps operators into action sequences. Secondly, the problem space is more complex and must accommodate some representation of the device which the problem solver is using. Hence, when writing a computer program using a text editor; the programming language constitutes the task language, and the commands used to manipulate the editor comprise the device language.

In previous chapters we have alluded to the fact that features of both the task and the device language can affect programming behaviour, and it would seem clear that any complete account of problem solving in programming must take these factors into consideration. The intention of the present chapter is to provide a review of studies which have examined the effects of task language features on programming behaviour and strategy. Unfortunately, there have been few studies which explicitly address the role of device languages in the determination of programming strategy, however a small number of studies have at least recognised their importance and have attempted to state clearly the contribution that features of the device language make to the performance of the programmers observed in these studies. Hence, the emphasis of this chapter is concerned with outlining features of task languages, and particularly their notational aspects, which have been shown to affect programming strategy and program comprehension.

Green (1990a) suggests that programming languages embody implicit theories of programming, and that language designs broadly reflect the concerns of these implicit theories. He suggests that three main stages can be identified in the development of implicit theories of programming. These stages reflect the historical evolution of programming languages and hence provide a useful framework within which to review empirical work which has attempted to explicate these implicit theories and to question their validity as accounts of programming behaviour.

5.2 Programming as errorless transcription

Green (1990a) characterises the first phase in the development of programming languages as embodying a theory which views programming tasks as involving an errorless transcription between some mental representation of a program and its code-based representation. He claims that the Fortran-BASIC tradition epitomises this view, and that these languages display a number of basic design features which reflect the errorless transcription approach:

"The Fortran programming system (punched cards) and the Basic line numbering system encouraged programmers to create their programs in the order of the text -i.e., line 1 of the final text was also the first line to be punched in. Thus, *the program was fully developed at the start of coding, needing only to be transcribed.*

Fortran and Basic have very few guards against typing errors, which can readily create a new text that is syntactically acceptable but not, of course, the intended program. By implication, *programmers do not make typing errors*.

Neither fortran nor Basic originally supported any use of perceptual cues to help indicate structure. Possible cues would have included indented FOR-loops, demarcated subroutines, bold face or capitals to indicate particular lexical classes, etc. The implication is that *programmers can comprehend the program text without assistance.*

The use of GOTOs as the sole method to determine control flow encourages small changes but makes large changes extremely tedious. The implication here is *programmers do not modify their first version except trivially*" (p. 123).

Hence, the errorless transcription view implies that a mental representation of a program should be regarded as a series of discrete steps and that programming involves translating each of these steps into its coded representation in the target programming language. Empirical research, mostly conducted in the 1970's, tended to focus upon a number of specific questions which arose from this view. For instance, a range of studies attempted to compare the use of GOTOs with nested conditionals (see the next section of this chapter for a review of these studies).

Other studies focused upon the individual syntactic constructions of various programming languages. For instance, Youngs (1974) analysed the errors made by novice and professional programmers according to the statement type where the error occurred (e.g., assignment, iteration, GOTO, conditional etc.). One of the more interesting results to emerge from Youngs' study was the large proportion of errors that were related to control statements (35% for novices and 51% for professional programmers). Youngs' collected data from a whole range of programming languages and so it is difficult to derive any specific conclusions which might inform language design. However, it is interesting to speculate about the extent to which the inflexibility of GOTO type control structures might have contributed to the errors found by Youngs.

5.3 The demonstrable correctness view of programming

The second phase of language development that is outlined in Green's review of implicit theories of programming is the so called 'demonstrable correctness' view of programming. This view is well represented by the structured programming school (Dahl, Dijkstra and Hoare, 1972; Yourdon, 1975) and by those languages which embody the principles advocated by this school, particularly Pascal. The main principle of the structured programming school is that programmers should work according to certain rules and within various constraints, and that this

disciplined approach will make it easier to prove that a program so constructed will be correct. This, of course, is a mathematical claim. However, in parallel, the structured programming school has made additional, and empirically testable claims, to the effect that the correct structuring of programs will not only lead to reduced errors but will also maximise the likelihood of program comprehension. Hence, structured programs will be easier to understand, to write and to debug. Here, comprehensibility is equated with structural simplicity and the idea that programs can be hierarchically constructed from various simple components.The structured programming school advocates two main principles.

The first principle is that programmers should use well-defined control structures. This principle led to the rejection of arbitrary GOTOs. It has been argued that if such structures were allowed, more information would be needed in order to characterise the progress of control flow in a program. For example, Dijkstra (1968) claims that "Our intellectual powers are rather geared to master static relations ... and our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our upmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible." Dijkstra then goes on to ask "Suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?". This is taken to suggest that the use of GOTOs to characterise control flow would mean that more information would be needed in order to understand the control flow of the program, and as a consequence, that programs constructed in such a way would be difficult to comprehend.

The second major principle of the structured programming school is that programmers should define and compose data structures and types in a hierarchical fashion. According to this view, the programmer is able implement the higher levels of a design and represent the lower levels by stubs which simulate their function in a simplified way. As the implementation of one level is completed, the programmer can then move on to a lower level in the hierarchy and implement that in terms of its sub-levels. Ultimately ,the lowest level in the hierarchy is implemented using basic programming language facilities. Note that

this principle, often called stepwise refinement, is also embodied in many models of human problem solving. Moreover, in the area of complexity theory, Simon (1962) has argued that systems which survive have three outstanding characteristics: they are organised in terms of a hierarchy of sub-systems, these sub-systems are often loosely coupled and the sub-systems themselves are highly internally cohesive.These three characteristics mirror precisely the main concerns of the structured programming school.

In language design, Pascal reflected these concerns by rejecting GOTOs and by instituting, instead, a small repertoire of nestable loops and conditional structures to describe control flow. In addition, Pascal enabled programmers to hierarchically define and compose data structures and types. Programs built using these principles should, if the claims of the structured programming school are correct, be easier to comprehend than unstructured programs. However, a second phase of empirical research, which attempted to address some of the claims of the structured programming school, indicted that structured programming does not necessarily constitute a panacea which can eliminate the problems that are typically faced by programmers.

5.3.1 Research on structured programming claims

For example, Sime, Arblaster and Green (1977) compared the performance of novices constructing small programs in three micro-languages (see also Sime, Green and Guest, 1973). Each of these micro-languages embodied a different control structure; a jump-if structure (jump), a nested begin else structure (nest-BE) and a nested if not end (nest-INE) structure (see figure 5.1). These three structures were used since they not only exhibit differences in nesting and jumping styles, but also differ in terms of the method used to indicate the scope of conditionals within each nesting style. Of these three notations, one (the Nest-BE notation) was structured, in that is used begin-end statements to mark conditional blocks in the normal way, one was unstructured (the jump notation - a GOTO like construct) and one was structured, as in the first case, but also contained additional information (the Nest-INE notation). This additional information was logically redundant since it could be derived from information already present in the program. Hence, in the example illustrated in Figure 5.1, the predicates

('juicy', etc.) are restated for the negative arm of the conditional ('end juicy'). The main hypothesis of this study, based initially upon the author's intuitions, was that this conditional restatement would improve performance.

Jump

```
    if hard goto L1
    if juicy goto L2
    chop roast stop
L2 fry stop
L3 boil stop
```

Nest-INE

```
if hard: boil
not hard:
    if juicy: fry
    not juicy: chop roast
    end juicy
end hard
```

Nest-BE

```
if hard then
    begin boil end
else
    begin
    if juicy then
        begin fry end
    else
        begin chop roast end
    end
```

Statement of problem:

Fry:	everything which is juicy but not hard
Boil:	everything which is hard
Chop and roast:	everything which is neither hard not juicy

Figure 5.1. The three forms of control structure and problem statement used in Sime et al (1977).

Sime et al showed that both of the nested conditionals led to improved performance (in terms of the number of errors in the resulting program) over the unstructured notation, however performance was best in the nest-INE condition where additional control information was provided. These results may provide some general support for the structured programming school, however it is not possible to delineate, in this experiment, the effects of structure from the effects of providing extra information. Sime et al go on to claim that enhanced performance in the nest-INE condition arises because the redundant repetition of predicates helps the programmer to 'deprogram', that is to translate the program back into the original problem statement.

Arblaster, Sime and Green (1979), took these ideas further by comparing the effects of structured vs unstructured notations on performance. In this study Arblaster et al, were interested in exploring the necessity of employing hierarchically structured notations, as implied by the structured programming school, as opposed to using notations which are structured in other ways. Their results demonstrated that several kinds of notational structuring (described as hierarchical, decision-table-like, and compromise) can improve program comprehension compared to a condition which used an unstructured notation. However, the hierarchically structured notation did not give rise to a significant improvement in performance compared to the other structured notations.

The results of the studies reported above raise some doubts about the central claims made by the structured programming school. In particular, it is not clear that structured conditionals improve performance in the way that would be predicted. Secondly, it also appears that other forms of structuring, in addition to hierarchical structuring, can lead to improved performance. However, these results must be interpreted with some caution. Firstly, a study conducted by Vessey and Weber (1984a) extended the three languages used by Sime et al to include indented and unindented forms of all three conditional structures. In their original experiment, Sime et al indented only the nested languages, since they claimed that the jump condition could not be indented without considerably restricting the syntax of the micro-language.

However, Vessey and Weber showed that the jump language could be indented with only a slight relaxation of the syntax (see figure 5.2). Their results demonstrated that the nested conditionals led to improved performance over the JUMP conditional, but only in their unindented forms. In the case of their indented forms, Vessey and Weber found little evidence in favour of nested languages over unstructured conditionals. Hence, it appears that subjects' performance in the tasks studied by Vessey and Weber was determined to a greater extent by indentation than was supposed by Sime et al, and this clearly raises some doubts about the relative advantages of nested conditionals over other conditional structures.

Indented modified JUMP

```
        IF hard GOTO L1
        GOTO L4
L1          IF green GOTO L2
            GOTO L3
L2          peel roast stop
L3          peel grill stop
L4          IF tall GOTO L5
            GOTO L6
L5          chop fry stop
L6              IF juicy GOTO L7
                GOTO L8
L7              boil stop
L8               roast stop
```

Figure 5.2. Vessey and Weber showed that the jump language used in the Sime et al (1977) study could be indented with only a slight relaxation of the syntax.

Another problem with many of these early studies of programming is that the questions they addressed were often inappropriate to the kinds of answers that such studies could provide. For instance, the primary concern of Sime et al was to evaluate the claim that nested conditionals are better than GOTOs. However, programming is clearly a complex skill, and it is quite probable that an advantage for nested conditionals might be evident in the context of certain programming tasks, but be absent in others, and that unstructured control forms might be more

useful in different task contexts (see also, Gilmore, 1990). This problem is also apparent in other studies, for instance, in Shneiderman's (1976) comparison of logical vs arithmetic IF statements in Fortran. The problem here is that research on micro-languages simply may not be generalisable to tasks other than the constrained problems employed in these studies. In addition, it is not clear whether the results of these studies will generalise to professional programmers, since they were concerned only with novices. However, Green (1977), has demonstrated that presenting professional programmers with hierarchically structured programs, together with extra information about the truth-values of particular variables could give rise to a significant speed advantage in terms of answering certain kinds of comprehension question.

Vessey and Weber (1984b) have produced a comprehensive review of empirical studies which have addressed structured programming claims, including those outlined above. They suggests that, in general, the evidence supporting structured programming is weak. However, according to Vessey and Weber, this not necessarily because the claims made by the structured programming school are wrong. Rather, they suggest that these problematic results are "a manifestation of poor theory, poor hypotheses and poor methodology". (pg 398). They go on to suggest that there are several specific reasons why previous results have turned out to be equivocal "First...the theory enunciating the effects of structured programming on software practice is rudimentary and inadequate; second.. this lack of a theory has inhibited the formation of hypotheses that contribute to both understanding and predictive powers; third..until the theory has been developed , it is not possible to identify the strategic hypotheses and, as a consequence carry out empirical research; and finally.. the existing empirical work reflects the shoddy state of the theory in that it does not effect a coordinated whole, nor has it aspired to understanding as opposed to prediction". (pg 406).

While the research reported above has not made a clear case either for or against structured programming, it has demonstrated that for certain tasks, and for certain groups of subjects, language features can and do affect programming behaviour. In general, we might suggest that the structured programming school has made claims which are either too extensive or too vague for proper systematic empirical study, and that empirical work has simply mirrored these claims by over generalising the significance of its results.

5.3.2 Extending work on language features to non-procedural paradigms

The work we have reviewed so far has been concerned with the effects of language features in the procedural language paradigm. However, in principle, this work should also apply to other paradigms, since most researchers who have studied notational design have been careful to phrase their explanations in terms of the general information processing requirements of particular tasks, and the extent to which specific notational features might support these requirements.

In this vein, Gilmore and Green (1984), attempted to test Green's (1977) assertion that the mental operations demanded by certain tasks are harder in some notations than in others by studying the effects of different notations on comprehension tasks. In particular, they were interested in testing the hypothesis that it would be easier to answer procedural-type questions than declarative questions, given a program with a procedural notation, and conversely, that it would be easier to answer declarative-type questions in comparison to procedural questions, given a declarative notation.

Gilmore and Green used four notations in their experiment. Two of these notations were procedural and two were declarative, and one of each pair contained cues to procedural or declarative information, respectively. One procedural notation resembled Pascal and contained cues to circumstantial information through the indentation of conditional statement, while the other resembled Basic, and used labels and GOTOs, instead of indentation (see Figure 5.3). Both of the declarative notations are described by Gilmore and Green as resembling production systems. In the declarative-uncued condition, the rules formed an ordered set , whose antecedent conditions were tested in order. In the cued-declarative condition, the rules again fired in order, but unlike the uncued condition, rules were displayed even if they contributed no action. Hence, in the uncued condition, control information is distributed piecemeal among the rules, whereas in the cued condition, all control information is immediately apparent.

```
    if both of ANN and JANE goto A
    if not SARH goto C
    SAIL (4)
C:  BIKE (1)
E:  if distance >= gangsize*4 goto D
    BUS (2)
    goto E
D:  if ANN goto B
    DRIVE (2)
    goto B
A:  WALK (4)
    if distance >= gangsize*4 goto B
    BUS (2)
    goto A
B:  BURY LOOT
```

a). Proc-No-Cues

```
begin
   if ANN and JANE then
      begin
         WALK (4);
         while distance<gangsize*4 do
         begin
            BUS (2)
            WALK (4)
         end
      end
      else
      begin
         if SARAH then SAIL (4);
         while distance < gangsize*4 do BUS (2);
      end
      BURY LOOT
end
```

b). Proc-Cues

P1: SARAH and at most one of ANN and JANE	SAIL (4) Turn P1 off	(A)SAIL (4) nothing	SARAH and at most one of ANN and JANE otherwise
P2: both ANN and JANE	WALK (4) Turn P2 off	(B)WALK (4) nothing	Both of ANN and JANE otherwsie
P3: at most one of ANN and JANE	BIKE (1) Turn P3 off	(C)BIKE (1) nothing	at most one of ANN and JANE otherwise
P4: distance < gangsize*4	BUS (2) Turn P2 on	(D)BUS(2), B, D nothing	distance<gangsize*4 otherwsie
P5: not ANN	DRIVE (2) Turn P5 off	(E)DRIVE (2) nothing	not ANN otherwise
P6:	BURY LOOT stop	(F)BURY LOOT.	

c). Decl-No-Cues

d). Decl-Cues

Figure 5.3. Examples of the programs used in Gilmore and Green (1984). The top pair of programs illustrate the procedural notations , with cues or without. The bottom pair of programs illustrate the two declarative languages.

Gilmore and Green asked their subjects to study the programs and then answer a series of questions which were intended to tap either sequential or circumstantial information.The sequential questions asked what either the next or the previous action in the program would be, given the occurrence of some event, while the circumstantial questions asked what condition would be true if a particular action either had or had not occurred. Subjects were tested in two situations, one in which they were able to consult the printed text of the program, and the other where they answered questions from memory.

Their findings provide general support for the idea that specific notations might be more suited to particular tasks. Hence, in general, their subjects were able to answer the declarative questions more accurately given a declarative notation, while the procedural notation facilitated correct responses to the procedural questions. However, these effects were weak, except in the condition where subjects were working from memory. Hence, the notational structure of the program appears to affect the ease with which information can be extracted from the printed page and also seems to facilitate recall. The first of these findings is perhaps not surprising, however second is rather more interesting, since it suggests that the mental representation of a program maintains some features of the original notation, and as a corollary that programs are not stored in a uniform mental language.

The idea that all programming languages are represented in some uniform mental language which preserves the semantics of the program but not its surface features, is implicit in many theories of programming (Soloway and Ehrlich, 1984; Shneiderman and Mayer, 1977). For instance, the frequent use of memory tests to assess program comprehension implicitly assumes that the same problem, represented in different notations, will give rise to the same memory representation. In support of this, Shneiderman (1977), cites Bransford and Franks (1971) who have demonstrated that in prose memory, syntax is rapidly forgotten, and that semantics are recalled.

5.3.3 Implications for plan-based accounts of programming behaviour

Moreover, the schema or plan-based account of program comprehension, similarly, relies upon the assumption that programs are represented in a generic semantic form, and that programming plans will be equally applicable regardless of the language being used. In chapter 3, we questioned the validity of this assumption, and there is now a significant amount of evidence which suggests that such plans are not language independent. For instance, Gilmore and Green (1988) have carried out a study which demonstrated that while plans may be useful to Pascal programmers, Basic programmers appear to rely to a much greater extent upon the control flow information which they are able to derive from the program. Hence, it appears that the plan theory may be able to account for program comprehension in the context of certain languages but not in others, and this may suggest that the plan theory is of significantly less utility that has previously been supposed.

Gilmore and Green (1988), examined the debugging behaviour of experienced programmers using two different languages - Basic and Pascal. Their subjects were presented with programs which had various structural elements highlighted. For instance, some programs had control structure highlighted. Here programs were indented in the normal way to provide a perceptual cue to control-flow. There have been many studies which have demonstrated the utility of indentation as a perceptual cue in the context of tasks that require an understanding of the control-flow structure of the program (Miara, Musselman, Navarro and Shneiderman, 1983; Kesler, Uram, Magarah-Abed, Fritsche, Amport and Dunsmore, 1984), and as such it can provide a useful perceptual cue without affecting the perception of other structures. Other programs used in the Gilmore and Green study were highlighted in a different way to provide cues to the program's plan structure. To achieve this, Gilmore and Green highlighted each separate plan in a different colour. The validity of using colour cues to indicate structure has been demonstrated independently by Van Laar (1989), who showed that colour can supplement indentation in showing control-flow in Pascal programs , with some net gain in performance for learners answering a variety of comprehension questions. Also, in addition to the two program formats outlined above, other programs were presented with no perceptual cues and others with both control-flow and plan structure cues.

The main intention of the Gilmore and Green study was to determine the way in which perceptual cues to different structures in a program can affect the performance of programmers in certain tasks. More specifically, they were interested in addressing three specific issues that arise from the plan-based view of programming. Firstly, they suggest that the existence of plans has mainly been inferred from protocol and error analysis, rather than from direct experimental evidence. They claim that direct experimental support for the plan-based view could be obtained by showing that perceptual cues to plan structure, such as the colour cues described above, improve the comprehensibility of programs. Secondly, they suggest that the plan theory would claim that plans constitute the primary mode of representation for the programming knowledge of experts and that plans represent the deep structure of the problem being solved. Gilmore and Green suggest that, as a corollary to this, providing a situation in which plans can be readily perceived should improve performance on all programming tasks. Finally, Gilmore and Green argue that all the existing evidence for plans had been obtained from studies of Pascal programmers, with the assumption that plan effects will generalise to other languages. By comparing the performance of programmers using different languages, Gilmore and Green were able to test this plan generalisability assumption.

In the Gilmore and Green study, these issues were addressed by comparing the detection rate and accuracy of their subjects in a debugging task for a variety of bug types, in situations where perceptual cues were provided to various program structures. Bug types were derived from one of four categories. Firstly, bugs could be described as surface level bugs, when they were independent from any particular structure in the program. Such bugs are likely to arise from typing errors and syntactic slips. For instance, surface level bugs might include missing or misplaced quotes or undeclared variables. Control flow bugs occur in the control flow of the program, but do not affect other structures, for instance, a missing begin statement. Plan structure bugs are, as the name implies, bugs which arise from the incorrect implementation of a plan and include such things as updating the wrong variable. Finally, there are bugs which are caused by structures interacting in the wrong way. For instance, both the control-flow structure and the plan structure of a program may be correct, but their interaction may contain errors; for example, initialisations within a main loop.

Gilmore and Green suggested four specific predictions which correspond to the predictions that they claim are implied by plan-based accounts of programming behaviour. Firstly, they suggested that the presence of cues will make no difference to the detection of surface errors. Secondly, that indentation cues to control flow will improve the detection of control flow errors. Thirdly, that the use of colour cues to indicate plan structure will improve the detection of plan errors, and finally, that the presence of both cue-types will improve the detection of interaction errors. Moreover, they argue that these effects should be evident in the case of both Basic and Pascal.

The main findings of this study pose a number of disturbing implications for plan-based accounts of programming, since while all the above hypotheses were supported for the Pascal data, none were supported by the data from the Basic programmers. The results of this study clearly demonstrate that plan structures are psychologically meaningful to Pascal programmers, since perceptual cues to plan-based structures give rise to an improvement in performance in plan-related tasks. However, plan structure cues did not improve performance in non-plan-related tasks, and Gilmore and Green suggest that because of this, plans do not represent the deep semantic structure of the problem. Finally, they claim that since plan structure cues do not enhance the performance of Basic programmers, it appears that such programmers do not view programs in the same way as Pascal programmers. While Pascal programmers may be influenced by plan structures, it appears that the behaviour of Basic programmers is influenced more by other structures, and in particular by control-flow.

Gilmore and Green interpret their findings as suggesting that certain notations may facilitate plan use, while different notations may more readily support the extraction and use of other sources of information. In particular, they advance the idea that some notations, and in particular Pascal, may be more role-expressive than others, and that role-expressiveness may facilitate plan use. Gilmore (1986) introduced the idea of role-expressiveness to describe a property of languages which facilitates the automisation of a mapping between the problem being solved and the programming knowledge that is brought to bear during problem solving. He suggests that there are three components to role-expressiveness; discriminability, statement-structure mapping and statement-task mapping.

Discriminability refers to the ease with which a notation provides access to chunks of code, where a chunk may be a single statement or a group of related statements. The initial step in perceiving the role of some statement is to differentiate it from other structures that surround it, and if this process can be aided then this will improve comprehension for certain kinds of task. Gilmore and Green claim that this process is facilitated in languages such as Pascal which display a rich set of lexical cues. Statement -structure mapping refers to the process of establishing the structural role of a statement, independently of the particular problem being solved. Gilmore and Green claim that establishing this mapping will be more difficult in unstructured languages such as Basic where the same piece of code may be used for more than one purpose (for example, as an initialisation or as an update. Statement-task mapping is a process which maps the structural role of statements to their task role. An example of the difficulties that can arise at this level is evidenced in Soloway, Boner and Ehrlich's (1983) observation that novice programmers who try to force a problem into a 'read-process' loop make errors when they are required to test the terminating condition twice.

The Gilmore and Green study clearly throws some doubt upon plan-based explanations of programming behaviour, and in particular upon the idea that plans are generalisable structures which are used during problem solving regardless of the programming language being used. In addition, this study points to some important notational features of programming languages which appear to underpin the ease with which certain structures, including plan structures, can be extracted from the program text. However, it appears that there are other explanations for the findings of this study which suggest other factors, in addition to notational features, which may have given rise to the results obtained.

For instance, in chapter 6 an experiment is reported which suggests that the learning experiences of Pascal and Basic programmers are often very different, and that this may have may have been a confounding factor and led to the differences in plan comprehension observed in the Gilmore and Green study. This experiment demonstrated that Basic programmers who had been taught structured programming techniques performed in a very similar manner to the Pascal programmers in the Gilmore and Green study, in terms of their ability to use cues to plan structure in plan related tasks. Conversely, other Basic programmers, who

were not taught structured programming, did not show an improvement in plan related tasks in the situation where plan structures were cued. In fact, as in the Gilmore and Green study, the performance of this second group was affected to a much greater extent by control-flow cues.

This suggests that the learning experience of programmers may contribute in certain ways to their comprehension and information seeking strategies, and that structured programming experience may in some way encourage programmers to focus upon a program's plan structure even though the notation of the programming language used may not facilitate plan use, as in the case of Basic. The results of this experiment are interpreted in terms of the broad theory presented in this thesis, whereby expertise is seen to be related, in part, to the restructuring of programming knowledge. This restructuring process may be facilitated by certain forms of programming experience, and especially by techniques such as structured programming which emphasise the functional role of certain programming constructs and encourage programmers to adopt decompositional strategies which suggest a hierarchical structuring of programming knowledge. Interpreted in this way, certain forms of programming behaviour are seen as dependent upon interactions between knowledge structures and notational features, and this suggests that the current polarisation between notational and plan-based accounts of programming behaviour may be inappropriate.

5.4 Programming as exploration

The third phase in Green's taxonomy of language paradigms and implicit programming theories reflects the present view of programming. According to Green, this view characterises programming as an evolutionary process in which various alternatives are explored and their consequences considered. In this way, programs are built from preexisting structures which are modified in certain ways according to the demands of the particular problem being solved. Recent language developments such as Smalltalk embody this evolutionary view of program development by encouraging code re-use and by emphasising the process of gradual incremental change in software development.

In chapter 4 we reviewed a number of studies which have attempted to characterise programming in this way. For instance, the parsing-gnisrap model of code generation proposed by Green et al (1987 a and b) and Gray and Anderson's analysis of change episodes have both emphasised the cyclical nature of programming activities. These models have promoted the idea that code is not generated in a linear fashion, but is instead generated in fragments - a facet of behaviour reflecting certain cognitive limitations - and that these fragments need to be continually reinterpreted and evaluated as coding progresses. In chapter 4 a detailed description of these models was presented, and this need not be repeated here. However, these models make certain predictions about the effects of notational features upon the nature of these cyclical coding activities, and it is to these that we now turn.

For instance, the behaviour of the parsing-gnisrap model is determined by a number of features, including the effects of certain notational properties of the language being used. In particular, Green et al suggest that the behaviour of this model is determined in large part by the ease with which already generated code fragments can be reinterpreted; that is, remapped back into an internal cognitive representation. They claim that this process will be facilitated if the language is role-expressive (see above), and in consequence that it supports the programmer's perception of the role of each program statement or group of related statements. Role-expressiveness, therefore will affect the parsing side of the parsing-gnisrap model. However, other language features will affect the generative aspect of coding behaviour.

In particular, languages which are resistant to local change (to use Green's terminology - viscous languages (See Green, 1989; 1990b)) will provide little support for non-linear generation. For instance, to insert a new line in Basic may involve renumbering existing lines and will possibly necessitate readdressing control structure assignments. In Pascal, adding a single statement or procedure may not be as difficult, however if this necessitates reconstructing the identifier hierarchy so that a given procedure can be brought into a lower block, then this may have many undesirable repercussions.

Viscosity is not only a property of task languages, but may also be manifest at the device level. For instance, some editors require much more effort to achieve a

given action, and a significant proportion of a programmer's behaviour may be taken up simply operating an editor. This is especially true when code is not generated in a linear fashion and where programmers have to navigate through the code. Such device viscosity is reflected in a study of CAD users conducted by Whitefield (1985), who found that users would often spend a significant proportion of their time simply operating the device and making no progress with the task in hand.

One principle question addressed in the Green et al study was the extent to which different languages might affect the distribution of non-linearities during code generation. Green et al analysed the behaviour of programmers using three languages -BASIC, Pascal and Prolog. These languages were chosen to exemplify some important differences in language design. Their results showed that although Pascal programmers produced programs with a very similar structure to the Basic programmers, the latter group generated code almost linearly, whereas the Pascal group engaged in many backward jumps to insert new material into already generated structures. Finally, the Prolog group fell somewhere between these two extremes in terms of the extent of non-linearities.

Green et al claim that their results provide evidence for predictions stemming from the parsing-gnisrap model, in particular that coding is fitful and sporadic and that the extent of this nonlinearity is dependent upon features of the language. They explain these results by suggesting that Basic is more viscous than Pascal, causing programmers to adopt a strategy which minimises interleaving, and secondly that Basic is less role-expressive than Pascal. Hence, Basic programmers will generate code in a linear fashion, since they would otherwise experience difficulty comprehending it. In the case of Prolog, Green et al are more conjectural about its notational properties, however they do suggest that it may rank low on the role-expressiveness dimension. In particular, they have observed that there are very few cues in Prolog which can be used to indicate the purpose of a specific piece of code. For instance, they suggest that it is impossible to know if a Prolog variable is to be used for input or output at any one time during the execution of a program.

5.4.1 Modelling the parsing of notations

More recently, Green and Borning (1990) have attempted to produce a computational model of notation parsing, based upon an extension of Kempen and Vosse's (1989) unification based natural language parser. Green and Borning refer to their model as a generalised unification parser, since it preserves Kempen and Vosse's general approach, but has increased power. In particular, it is capable of using typographical features to aid the parsing process.

Their parsing model is programmed with an elementary grammar of procedural language cliches or plans (see figure 5.4). For instance, text patterns representing counter plans start with an initialisation statement, in which some identifier is set to an exceptional value, commonly zero. Secondly, there will be a statement corresponding to a start of loop, followed by an add-1 statement, where an identifier is set to itself plus one. Finally there will be an end of loop which completes the counter pattern. Other statements may of course intervene between these various plan components, however these are ignored during parsing[1].

Green and Borning have extended the Kempen and Vosse parser in various ways and in the present context, two of these extensions are of particular relevance. Firstly, their model can use display-based features to aid the parsing process. Hence, the salience of a particular statement and its corresponding level of activation in the parse space can be affected by its indentation level, by its colour coding (i.e., as in the Gilmore and Green (1988) study) or by other typographical features.

Secondly, in the Green and Borning model, beacon constructions (Wiedenbeck, 1986 a and b, and see chapter 4) are assigned permanently raised activation values in the lexicon. Green and Borning do not discuss how beacons are identified, however they suggest that raising their activation level will have the effect of making the model search for beacons. They suggest that an experienced programmer will know which constructions are likely to be the most efficient ones to parse first (See Brooks, 1983 and chapter 4), and hence their model captures this salient behavioral feature of the programming activity.

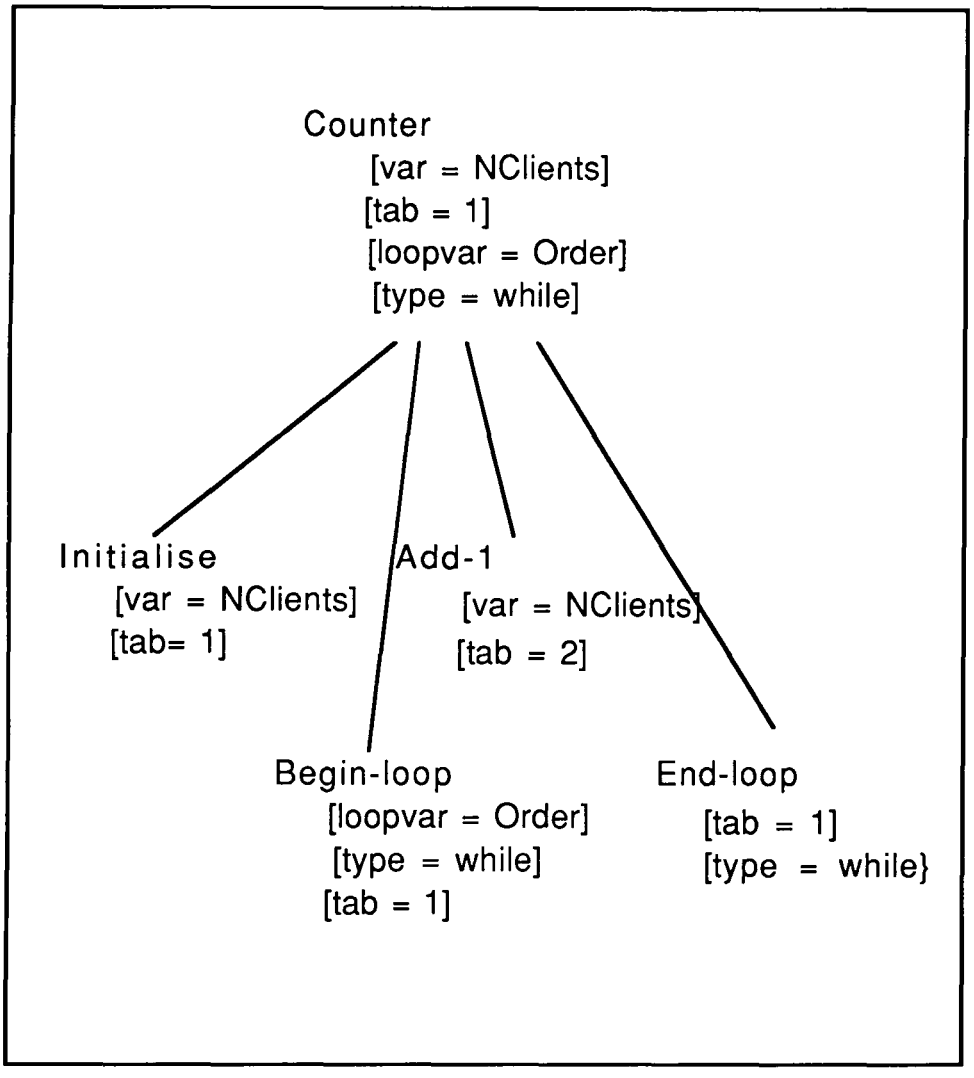


Figure 5.4. The elementary grammar of procedural language cliches or plans used in Green and Borning's parser.

Their model makes a number of comparative predictions about the ease with which various notations can be parsed. In general, parsing difficulty will increase when elements in the parse tree, or unification space, have similar activation levels. In the context of this model, the probability of unifying any two elements in the parse tree depends primarily upon their activation value. Hence, when a programming language has few lexical or typographical cues, the parser's ability to differentiate

structure will decrease and parsing difficulty will increase correspondingly. For example, they suggest that their model would predict that Prolog would be a difficult language to parse since it has no conventions for indentation and has fewer distinctive typographical constructions than many other languages.

Green and Borning report that they are attempting to extend their parser to include an elementary grammar of Prolog cliches. If their intuitions about the difficulty of parsing Prolog are correct, then this is likely to have significant implications for both the teaching of Prolog and for associated programming environments. Prolog has developed a reputation for being a difficult language to learn, and studies of Prolog learning have suggested that this may be because of the difficulties learners experience with its underlying conceptual model (Ormerod, Manktelow, Robson and Steward, 1986; Ormerod, Manktelow, Steward and Robson; 1990; White, 1987). In particular, one difficulty arises in determining the execution path from the program text, and to ameliorate this, systems which have incorporated impressive animations of Prolog execution have been developed (Brayshaw and Eisenstadt, 1988; 1989).

However, as Green and Borning suggest " the difficulty of parsing Prolog correctly may have contributed to the difficulty experienced by novices. A corollary of our work would be that a Prolog parser which picked out and labelled familiar cliches could significantly assist the learner." (pg 956). This would parallel work in other languages such as Van Laar's (1989) colour coded Pascal environment, and Green and Cornah's (1984) 'Programmer's Torch' which was designed to highlight cliche structure in Basic. In addition, the Green and Borning study draws into question the importance of execution animation over other techniques intended to improve program comprehension, and in particular techniques which illuminate a program's plan or cliche structure.

5.5 Conclusions

The work reviewed in this chapter has demonstrated the effects that certain notational properties of programming languages can have on both the nature of program generation strategy and upon comprehension success. This work has primarily addressed two themes.

Much of the early work on notational properties was concerned with evaluating the claims made by the structured programming school and, in particular, the idea that certain conditional forms will be more comprehensible than others. While this exercise has clearly proved to be useful in terms of questioning the basic assumptions of this paradigm, it is less clear whether these findings can inform notational design outside the narrow class of problems and restricted micro-languages that have been studied. In addition, it is questionable, whether the results of these studies are generalisable to more complex programming tasks.

The second phase of research into notational properties has been concerned more explicitly with examining the way in which notations can support the cyclical generation/evaluation activates that have frequently been observed in the context of programming studies. For instance, this work has addressed the way in which certain notational properties can affect generation strategy by demonstrating the extent to which such properties might support nonlinear generation. In addition, this work has suggested that comprehension success, at least in the context of certain tasks, is also affected by the notational properties of the language that is being used. In particular, it has been shown that task language features can influence the ease with which a program text can be decomposed or parsed into its constituent plan structures. This second phase of work has led to the description of various notational dimensions (Green, 1989) which, it is claimed "apply to many types of language ... and control how (or whether) the preferred cognitive strategy for design-like tasks can be adopted." (Green, 1989, pg 443).

In the context of the restructuring model presented in this thesis there appear to be at least three primary notational properties or dimensions that are of relevance. This model relies upon the notion of focal expansion to explain how programs are generated and adopts the basic principle of the parsing-gnisrap model which emphasises the fragmentary nature of code generation and evaluation. Hence, the ease with which focal structures can be parsed back into plans will be likely to affect both program generation strategy and comprehension success. This will depend upon the extent to which a particular language can be described as role-expressive. In addition, since this restructuring process is proposed to underpin the development of expertise, then there should be an interaction between expertise and language used, where the languages are here distinguished by the extent of their role-expessiveness.

Secondly, the extent to which plan structures (as represented in code) are contiguously distributed, in a spatial sense, will also affect the success of focal expansion. Hence, in languages where plan structures are diffuse, the programmer will have to do more work at the device level in order to implement a plan. In addition, as we have seen, languages which enable programmers to implement a base case or initialisation adjacent to the main procedure may facilitate comprehension. In this situation it is not necessary for programmers to link together various spatially disparate areas of a program during comprehension in order to recreate the program's original plan structure.

Finally, and related to this, the viscosity of the language will also affect the success of focal expansion. The model of knowledge restructuring presented in this thesis suggests that focal lines represent a discrete level of design abstraction. In addition, the model claims that focal lines will tend to be generated first during coding, following a hierarchically levelled approach to design decomposition. Hence, if features of the language make it difficult to insert subsidiary plan elements then the focal expansion process will be disrupted, and this disruption will result in the adoption of different forms of generation strategy.

Notes

1. Rich and Wills (1990) present an alternative program parsing method based upon a graph parsing technique which takes a program text as input and produces plan cliches as output. Rich and Wills claim that this output can be used to reconstruct a program's design and automatically generate documentation.

Chapter 6. The effects of the possession of design skills upon the perception and use of programming plans.

6.1 Introduction

In chapter 3 we introduced the notion of the programming plan. It was suggested there that descriptions of programming behaviour that have been couched in terms of plan theory have provided reasonably successful accounts of novice and expert programming behaviour in certain kinds of experimental task. However, more recently, the primary claims of the plan theory have been brought into question. In particular, subsequent experimental work has questioned the ability of the plan theory to account for the existence and the use of plan structures in languages other than Pascal. In addition, other work has thrown some doubt upon the relationship between programming plans and the development of expertise in programming.

This chapter reports two experiments that address these issues in further detail. The first experiment explores the extent to which plan knowledge guides the debugging behaviour of experienced programmers. These programmers had equivalent levels of experience with the programming language used in this study (Basic), however half of this experimental group had received design training. The second experiment looks at the recall of programs by design experienced and non-design experienced subjects. The intention of these experiments was to examine the extent to which the use of programming plans might be generalisable to languages other than Pascal and to explore the role played by design skills in the development of such plans. This allows us to examine the two central claims of the plan theory. Firstly, that programming plans are universal structures used by experienced programmers, despite the language being used. Secondly, that plans are the defining characteristic of programming expertise.

6.1.1 Programming plans and expertise

Soloway and Ehrlich (1984) present a fairly straightforward view of the relationship between plans and expertise. They suggest that experts possess and use appropriate plan structures while generating and comprehending programs and that novices typically do not. However, little is said about the processes involved in becoming an expert. Rist (1985) suggests that expert programmers develop both more plans and plans at a higher level than novices, and that the development of plans is characteristic of programming expertise. Another possibility is that novices may in fact possess an extensive range of plan structures but have difficulty mapping these structures onto structures in the target programming language.

Gilmore and Green (1988) claim that the plan theory of programming implies that programming plans represent the "underlying deep structure of the programming problem" (p. 423). Hence, if novices understand at least some aspect of the problem domain, as one might reasonably expect in certain cases, then it would be possible to assert that plans exist, and that they represent structures in that domain, but that novices have to learn how to express these plans in a particular programming language. Knowing that an average is a sum divided by a count must clearly be some form of 'natural plan' which would presumably be observed in non-programmers. Hence, novices must possess such knowledge if they are familiar with the problem domain. One of the limitations of the plan theory of programming is that little is known about the sorts of factors that might be involved in the development of the ability to map these 'natural plans' onto code structures.

6.1.2 The generalisability of programming plans

The second major problem with the plan theory of programming is concerned with the generalisability of plans to languages other than Pascal. Soloway and Ehrlich claim that programming plans are one of the major components of programming expertise.

However, Gilmore and Green (1988), have recently drawn into question the generality of the programming plan as a description of the main type of representation employed by the expert programmer (see chapters 3 and 5). Gilmore and Green suggest that the notation of certain programming languages may make those languages amenable or otherwise to the identification and use of plans. This suggestion is based upon the finding that Basic programmers are unable to benefit from cues to plan structure, while de-bugging programs, in the way that Pascal programmers are.

They conclude that Basic programmers do not appear to employ an abstract plan-based representation of a particular program while attempting to understand that program, but rely more extensively upon the control flow information embedded in the notation. Gilmore and Green suggest that Basic is less "role expressive" than other languages. That is that Basic programs are less discriminable from each other than are, say Pascal programs. In the case of Pascal, they argue that features of the notation of the language, and in particular its role expressiveness, make it easier for the programmer to infer the role of a particular statement and to discover the relationship between groups of statements (see chapter 5).

This work has a number of implications. Firstly, it suggests that the programming plan may not be a universal construct that is common to all programming languages (most previous studies on programming plans have been concerned only with Pascal or very similar languages). Secondly, that where plan structures do exist, they may not constitute the *exclusive* nor even the primary source of information relevant to program comprehension. Problems of this nature challenge the fundamental theoretical suppositions which underlie the notion of the programming plan and thus create something of an impasse.

To sum up, previous research appears to suggest two potentially divergent views;

a) that programming plans are universal natural strategies that characterise the cognitive representation of a program and the programming activity of the expert programmer and that the existence of plans can be taken to be a reflection of this expertise. Such plans are thought to represent the 'deep structure' of the

programming problem. This approach might be termed the 'Plans as Natural Artifacts view'.

b) that programming plans might be best regarded, in some circumstances at least, as artifacts both of a particular language and the structure that this language imposes on the programmer via the constraints of its specific notation. This approach might be called the 'Plans as Notational Artifacts view'.

The two experiments reported in this chapter present a third view on the nature of programming plans. Experimental evidence is cited that provides the basis for an alternative interpretation to present views, and suggests, in addition, what might be regarded as a more parsimonious and consistent analysis of existing experimental data.

6.1.3 The relationship between design experience and programming plans

The experimental work reported here suggests that both the above interpretations of the nature of plans maybe incomplete and that programming plans might be more suitably characterised in terms of their specific relationship to the way in which programming is taught. It appears that such plans cannot be regarded exclusively as natural structures that have evolved independently of learning about a language nor can they be considered solely as static properties of a program i.e. as mere artifacts of the structure that a particular language might impose.

The rationale underlying this alternative view on the nature of programming plans relates to two factors. Firstly, the way in which the differential learning experience of programmers may be reflected in the type of programming language used by a specific population of programmers and secondly, the effect that this may have on the development and use of programming plans.

Many experienced programmers will have learnt to program within the context of a formal course in which programming itself formed only a part. Programming is often taught in conjunction with the development of program design skills. For example, major methodologies such as Jackson Structured programming

(Jackson, 1975) or one of the more important design notations such as structure charts (Constantine and Yourdon, 1979) and design description languages (Chu, 1978). Such skills are intended by their nature to be independent of the particular programming language employed.

Despite the increasing emphasis placed upon the teaching of program design skills it is clear that many programmers are taught to program without the benefit of any formal training in program design. Programming is often taught in isolation as an adjunct to other subjects (engineering, business etc.). Moreover, it can be seen that the type of programming language taught to those groups who also learn about design skills, is often radically different than that taught to those who learn a language in isolation. Hence one might expect the so called structured languages, Pascal, Algol, C and the like to be the mainstays of courses associated with the teaching of formal design methods.

Conversely, Basic often predominates as a general purpose language in groups where programming is used to support other activities. Interestingly, this dichotomy is also reflected in the differences found in more objective classifications of programming languages by usage where Pascal, C and Algol fall into a clearly differentiated group of languages,while Basic and Cobol fall into another (Doyle and Stretch, 1987).

Previous experimental studies investigating the nature of programming plans across languages have ignored the fact that groups of programmers experienced in using different languages, despite exhibiting similar levels of programming competence, may have been exposed to widely differing kinds of backgrounds. Similarly, studies examining the nature of programming plans within a single language, have ignored the possible effects of the teaching of design skills on the development and use of such plans.

It would perhaps be unreasonable to suggest that the proponents of the natural artifact view of programming plans would rule out the possible effect of teaching on the development of plans. However, the plan theory suggests that plans constitute the expert programmer's mental representation of a program and consequently that they (plans) represent the deep structure of the problem.

However, if the development of plan structures is one of the major defining characteristics of expertise, then differences in teaching strategy per se, should not affect the nature of that expertise.

Most design methodologies place an emphasis upon problem decomposition and this may facilitate the ability to perceive common methods of solution within and between problems. Hence, it might be claimed that 'natural plans' exist, and exist within the problem one is trying to solve, but that novices have some difficulty expressing these in the target programming language. In addition, these plans or features of these plans need not correspond, except perhaps coincidentally, with the sorts of structures that arise as a product of the design activity. However, 'design skills' might facilitate the ability to perceive features of commonality between those structures which are the products of design, and are not expressible directly in a programming language, and 'natural plans' which in turn form useful structures that can be expressed in a programming language.

An investigation into the role of the effects of the teaching of program design on the existence and use of programming plans has a number of important implications for our understanding of such plans. If programming plans are to be regarded as natural artifacts that represent universal cognitive strategies which in some way facilitate the activity of programming by providing the basis for a cognitive representation of a program, then one would expect experienced programmers, whatever their background, to possess and employ such plans while generating or attempting to understand a program. If this is not the case then we must clearly rethink our theoretical position on the nature of programming plans, because of their assumed universality.

Denying this universality would mean that proponents of the Natural Artifacts view would have to make what might constitute possibly unacceptable concessions to their theory in order that it remain consistent. Indeed, if the plan theory is not generalisable to languages other than Pascal then clearly it of significantly less utility and therefore interest. If one is to view programming plans as Notational Artifacts then one might expect certain programming languages to facilitate (eg Pascal) or to discourage (eg Basic) the perception of plan-structures in that language regardless of the programmer's particular background in design.

Of course, if such design courses teach plans or how to find and use them, then it would not be surprising in the least to discover that those who have taken such a course use plans with more success than those who have not. However, the design courses discussed in this chapter were concerned exclusively with the teaching of structured design methods (in particular, Jackson structured design) and functional decomposition, and not with the explicit teaching of the sorts of plan structures identified by the Soloway group. Indeed, the exercises associated with these courses required students only to produce program designs and not implementations of these designs in any target programming language.

The experiments reported in this chapter address the hypothesis that programming plans can be at least partially characterised as artifacts of design or, as artifacts of the teaching of particular program design strategies. Taken as a whole these experiments do not assume homogeneity of experience between subject groups, rather groups have been chosen precisely because their backgrounds differ in terms of the level of design experience possessed by the groups. It must be noted however that as far as their ability to generate correct programs and to de-bug programs is concerned all groups exhibited equivalent overall levels of programming competence. It is interesting to note that most previous studies which have examined the role of programming plans in expert programming performance have attempted to measure performance factors that relate to either the recall or the generation of plan-like structures in programs. Hence programming plans are used to both explain the difference between novice and expert performance whilst at the same time providing the only measure of that performance. This lack of any independent means of evaluating programming performance seems to have clouded the theoretical interpretation of such research.

Two experiments are reported each looking at a separate aspect of the development and the nature and role of programming plans. The first experiment is concerned with the effects of the cueing of salient information structures in Basic programs and the effect that this has on de-bugging for design/ non-design experienced programmers. The second experiment examines the recall of plan structures by design and non-design experienced programmers.

The first experiment follows quite closely a design presented by Gilmore and Green (1988), in which Basic and Pascal programmers were asked to find bugs in programs in which various information structures were highlighted (see chapter 5). Gilmore and Green found that the highlighting of plan structures was advantageous to experienced Pascal programmers, but of little benefit to Basic programmers who appear to rely more extensively on control based information in programs. They conclude that the notation of Basic is less 'role expressive', making the identification and use of plans more difficult.

However, another interpretation of these results might be to suggest that Pascal programmers are more likely to have access to design based skills than Basic programmers, and that these skills facilitate the identification and use of plan-based information in programs. Indeed, Gilmore and Green point out that the Basic programmers used their experiment were engineering students while the Pascal programmers were computer science students. It would not be unreasonable to assume that the latter group had some experience of program design as most computer science courses now have a program design component. In the case of the former group this assumption is less valid since Basic is often taught in isolation to the teaching of design skills. Hence, evidence that is put forward for the effects of notation on the comprehension of plan structures in programs might equally well be interpreted as arising from non-trivial differences between the subject groups.

In fact, Gilmore and Green acknowledge that notational factors and in particular 'role expressiveness' might not be the only factors that influence the use of plan-structures in different programming languages. Indeed, they argue that different teaching strategies may provide a different emphases to the way in which such structures are perceived in programs. However they say little more about this issue except to say that more research is required into the influence of educational factors in determining the development of expertise in programming.

If one considers the "natural artifacts" view of programming plans, as we have characterised it, then one would expect no significant or systematic differences to exist between groups of design/non-design skilled programmers in terms of the benefit of the provision of cues to plan structure as compared to other types of

cue. Indeed in the experiment reported by Gilmore and Green, both groups exhibited similar overall levels of programming competence (as measured in terms of their general ability to debug programs). Hence, if programming plans are the major defining characteristic of programming expertise, then cues to plan structures in Basic programs should help the expert Basic programmer to detect plan-related bugs in the same way that cues to plan structures in Pascal programs should aid the expert Pascal programmer in the detection of plan-related bugs. However this hypothesis is not supported by the results of the Gilmore and Green study.

The first experiment reported in this chapter has been carried out in order to investigate the effects of the possession of design skills on the perception of cues to plan-structures in Basic programs. If the notational features of a language, and in particular its role expressiveness, are the primary factors that determine whether plan-structures can be usefully employed in the comprehension of programs written in that language, then cues to plan structure in Basic programs (which offer low role expressiveness) should be less useful than cues to other information structures, and in particular cues to control-based information. However, if the hypothesis that the possession of design skills facilitates and enables the programmer to both perceive and use plan structures in programs is correct, then cues to plan-structure in Basic should aid the design skilled programmer more than the non-design skilled programmer. In addition, if we assume that both of these groups are equally competent with the language, then the overall detection and correction of bugs by both groups should be broadly comparable. If this is the case and design skilled programmers benefit more than non-design skilled programmers from the provision of cues to plan structure, then the notion of the programming plan cannot provide the basis for a mechanism that can straightforwardly explain the nature of expertise in programming.

The second experiment reported in this chapter is a longitudinal study which is concerned with the effects that the teaching of design has on the recall of plan-based structures in Basic programs. Looking at the accuracy, speed and the order of the recall of programs provides us with the opportunity to examine the possible relationships between design experience and the perception and comprehension of programs.

Soloway and Ehrlich (1984) have analysed the recall of plan and non-plan-based Algol programs in order to test their hypothesis that if plan-based structures in programs facilitate the comprehension of those programs, then the recall of the salient features of plan-based programs is likely to be achieved more quickly and with greater accuracy than that of non-plan-based programs (see chapter 3 for a more detailed discussion of these experiments).

According to Soloway and Ehrlich, if programming plans help programmers to encode a program more efficiently, then expert programmers should recall first those lines (the critical lines) that make programs plan-like before they recall other elements of the program.

Gilmore and Green (1984), again using a program recall paradigm, have examined the notion that all programming languages are translated into a single type of cognitive representation when they are encoded by the programmer. Positing a single, universal, representation of programming knowledge has a strong affinity with the ideas which underlie the approach that we have characterised as the Natural Artifact view of programming plans. In this study Gilmore and Green explored the way in which individuals reproduce aspects of a specific language notation when they are asked to recall a program. The results of this study are used by the authors to illustrate their contention that the mental or cognitive representation of a program maintains certain salient features of the original notation of that program and as a corollary that the representation of the comprehended version of the program is not stored in a uniform "mental language" that is in some way independent of its external form.

Again we appear to have encountered something of a potential dichotomy between the natural and notational views of programming plans. On the one hand, plans are proposed as mechanisms through which one can explain the internal cognitive structures which underlie the mental representation of programs. On the other hand, features of the mental representations of programs (Gilmore and Green do not mention plans explicitly) are regarded as notational artifacts that are derived from the external structure of the program.

If we are to claim that design experience (rather than mere programming experience alone) facilitates the ability of programmers to use plan-structures in the comprehension of programs then we would expect the following hypotheses to be supported by the findings of the second experiment reported in this chapter.

1). During the first trial there should not be a significant interaction between Program type (plan-like or unplan-like) and Group (design skilled/non-design skilled).

2). During the second trial this interaction should be significant.

If these hypotheses are supported we will be in a position to claim firstly, that after learning about design, programmers are able to recall plan-structures more effectively than before. Secondly, this effect will have been demonstrated to be independent of mere programming experience, since both groups attended the same Basic programming course and had presumably attained the same level of competence in Basic. The conventional interpretation of the programming plan would not be able to account straightforwardly for this finding. If programming plans are the major characteristic of programming expertise then programmers of equal competence -with the same level of exposure to the language- should demonstrate a similar level of plan recall. This perhaps highlights again one of the fundamental problems of research into programming plans. That is that the notion of the programming plan is used to both explain the nature of expertise and provide the only measure of that expertise.

6.2 Experiment 1, the effects of cues to program structures

6.2.1 METHOD

Subjects

A total of 72 students participated in the experiment. One group of 36 subjects was drawn from a population of computer science undergraduates all of whom had attended a course on program design (Group A). A second group of equal size

was recruited from courses in finance, accountancy and engineering. This group (Group B) had no experience of program design. Both groups had an equivalent level of Basic programming experience amounting to at least 18 months. The design courses discussed in this chapter took place over a period of two academic terms. During the first term students were instructed in the underlying philosophy of structured design and functional decomposition. The second term was given over to the application of these techniques, and students participated in design exercises for which feedback was provided but no assessment made. Students were not expected to produce implementations of their designs in a target programming language.

Materials

The experimental materials used in this study consisted of three versions of programs written in Basic. These programs were based upon a program intended to calculate average rainfall (Johnson and Soloway, 1985). Five versions of a first (practice) program were created and 10 versions of the remaining two. All programs contained two bugs drawn from the Yale Bug Catalogue 1 (Johnson, Soloway, Cutler and Draper, 1983).

These bugs were of three types and were evenly distributed between the programs with a maximum of two in each. Ten bugs were unrelated to any particular code structure (for example, incorrect operator in arithmetic calculation, typographical error etc.), ten were related to control structure (for example, incorrect line-number assignment in GOTO statements) and a third class of ten bugs were related to the plan structure of the program (for example, no guard for invalid input, updating of wrong variable etc.).

Three representations of each of these programs were established. One representation provided no cues, a second used indentation in the normal way to reflect the underlying control structures of the program and the third used colour to indicate plan structure. In the case of the latter representation, lines of code which belonged to the same plan were indicated by presenting them on the screen grouped in terms of a particular colour. This representation did not make use of indentation.

The plan structures used were Input plan with guard, Running total loop plan, Counter variable plan and Guarded counter variable plan (Johnson and Soloway, 1985).

Figure 6.1 shows an example of one of the programs used in the experiment. Figure 6.1a shows the correct version of the program with plan-structures highlighted. Figure 6.1b illustrates a program which contains a number of errors.

```
10 REM avrprob
20 LET count = 0
30 LET Sum = 0
40 REPEAT
50     INPUT New
60       IF New = 99999 THEN GOTO 90
70           LET Sum = Sum + New
80             LET Count = Count + 1
90 UNTIL New = 99999
100 IF Count = 0 THEN PRINT "No legal inputs" ELSE PRINT
                                "Average  is..... "; Sum/Count
110 END
```

Figure 6.1a). correct version.

```
10 REM avrprob
20 LET count = 0
30 LET Sum = 0
40   REPEAT
50       INPUT New
60         IF New = 99999 THEN GOTO 100
70             LET Sum = Sum + Count
80               LET Count = Count + 1
90   UNTIL New = 99999
100 IF Count = 0 THEN PRINT "No legal inputs" ELSE PRINT
                                "Average  is..... "; Sun/Count
110 END
```

Figure 6.1b). Program with control flow error line 60, plan error line 70, and surface error line 100.

Figure 6.1. An example of a Program used in the first experiment. Different fonts represent the colour highlighting of different plans. Lines 20 and 80 - Counter variable plan. Lines 50 and 70 - Running total loop plan.

Design

The experiment was a three factor design. The independent variables were:

1). The type of structural cue provided - No Cue, Control Cue or Plan Cue.

2). The level of design experience of the group - Group A, design experience and Group B, no design experience.

3). Bug-type - Surface, control or plan.

Two factors (1) and (2) were between subjects factors; factor (3) was a within-subjects factor.

The dependent variable was the number of errors detected and corrected in a limited, fixed amount of time.

Procedure

After a short practice session, in which subjects were given feedback relating to their performance, the experimental programs were presented to subjects at random on the screen of a microcomputer. Subjects were instructed to attempt to find errors in the programs and to highlight these on the screen using a light pen. Transcripts of this activity were obtained. Subjects were given a natural language specification of the problem (from Johnson and Soloway, 1985) and allowed 1.4 minutes to locate the bugs in the programs. Subjects were explicitly told that each program contained only two bugs. They were then asked to correct these errors using a familiar screen editor and were allowed 5 minutes to complete this activity. Again, transcripts of this editing activity were obtained for later analysis.

6.2.2 Results

The results of this experiment, represented graphically in figures 6.2 and 6.3, were analysed using a three-way analysis of variance.

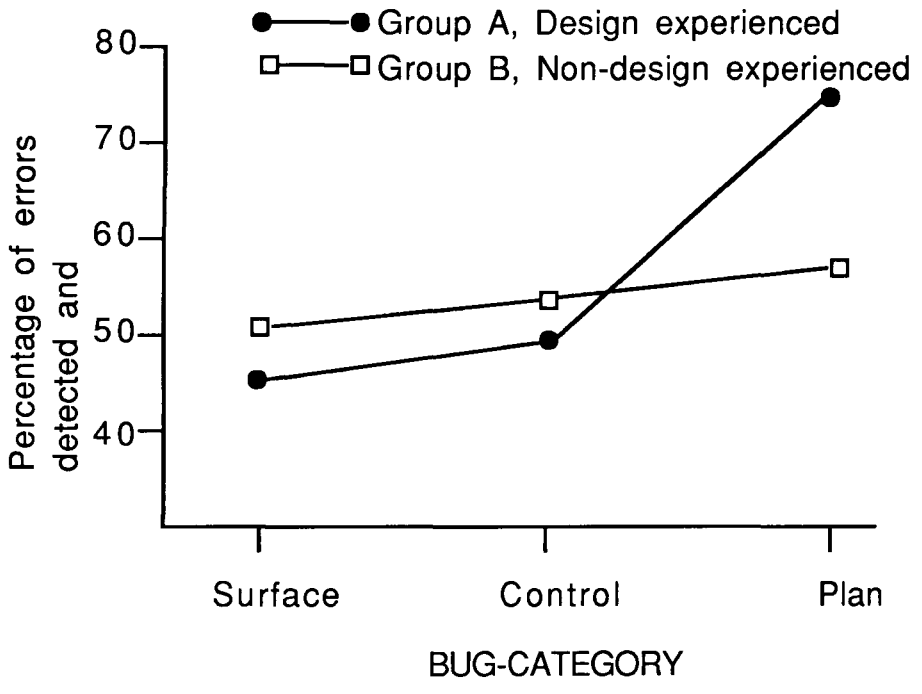


Figure 6.2. The location and correction of errors by design experience and bug-category.

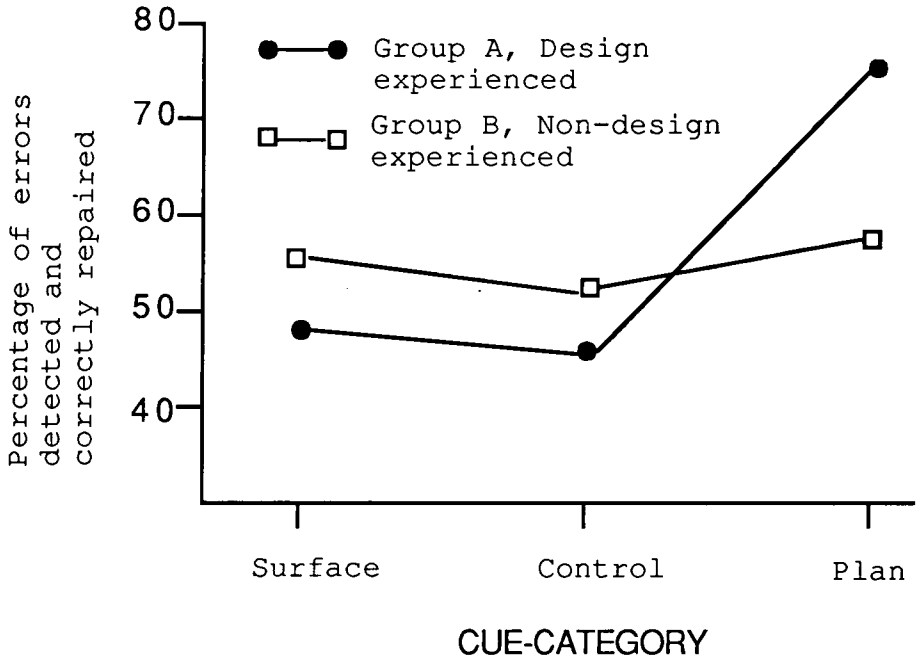


Figure 6.3 The location and correction of errors by design experience and cue-category

This analysis revealed no main effect of design experience, bug-type or cue-type. Hence design experience does not appear to improve the programmer's overall ability to detect bugs or to use cues. In addition, there was no three-way interaction between design skill, cue-type and bug-type. Such an interaction might be expected if the possession of design experience impacted upon the programmer's general ability to use cues to structure or to detect bugs. Interactions between design skill and bug type and design skill and cue type were evident. The design skill x bug type interaction was significant ($F_{2,30} = 14.5$, $p < 0.01$) as was the design skill x cue type interaction ($F_{2,30} = 18.6$, $p < 0.01$).

This suggests that design experience does have a significant impact upon a programmer's ability to detect plan-related bugs and use cues to plan structure. A cursory examination of Figures 6.3 and 6.4 suggests that while design skilled subjects are able to use plan related cues and to detect plan related bugs, these abilities appear to prevail at the expense of an ability to use other types of cue or to

detect other types of bugs. Indeed, a significant interaction ($F_{1,15} = 6.86$, $p < 0.05$) is evident between design skill, bug category and cue category (omitting plan bugs and plan cues).

6.2.3 Discussion

These results clearly demonstrate the effect of the possession of design related skills on the comprehension of plan-based structures in programs. In general, subjects in both groups demonstrated a similar level of programming competence. Hence, no main effect on the detection and correction of errors by design/non design experienced subjects was found. In fact, the overall percentage rate of error detection and repair only differed by two percent between groups for both cue and bug category (Bug category: Group A, 56%, Group B, 54%. Cue category: Group A, 57%, Group B, 55%). However, the detection and correction of errors by the design experienced group increased significantly when plan-structures were highlighted or when the errors in question were related to one of these plan-structures (see Figures 6.3 and 6.4). This is reflected in the interactional effects that were found to exist between the possession of design experience (Group) and bug category and design experience (Group) and cue category.

Contrary to the results of Gilmore and Green (1988), Basic programmers do appear to benefit from the provision of cues to plan structure when attempting to locate and repair errors in programs. However, it must be noted that this effect is only significant for the group that possessed design experience. From their study, Gilmore and Green suggest that the failure of Basic programmers to comprehend plan structures in programs is a reflection of the strictures of the notation of Basic; in particular, that it is less role expressive than Pascal. However, the failure of Basic programmers to benefit from cues to plan structure in comparison to Pascal programmers might simply be a reflection of the differential design experience possessed by each group.

Advocating the concept of programming plans to explain the difference between novice and expert performance in program generation and comprehension would therefore appear to be too simple a view. On the basis of the experiment reported

above, there does not appear to be a significant degree of uniformity between the use of plans by groups of programmers who would be regarded, on the basis of their ability to detect and repair errors, to possess very similar levels of programming competence. The results of the above experiment challenge both the natural and the notational artifact views on programming plans. Such plans appear not to be universal natural strategies or constructs that characterise or reflect the expertise of the programmer nor can they be regarded merely as notational artifacts that are imposed by the constraints of the particular structure of the language in question. Plan-structures appear only to provide the basis for the comprehension of programs for those programmers trained in design. In addition, plan structures are employed in the comprehension of Basic programs despite previous suggestions that Basic, because of features of its notation, does not facilitate the development and use of plan-like structures.

Another interesting result is that the ability possessed by design experienced programmers to detect plan-related bugs and use cues to plan structures appears to hamper their ability to detect control and surface bugs and to make use of control and surface cues. In other words design experience might be seen to focus attention on plan-like bugs/cues at the expense of other possible sources of bugs/cues. Without more research we are not in a position to comment on the possibility that in actual programming practice more surface and control bugs may infest programs than plan bugs. If this were the case we might be forced to adopt the rather disturbing conclusion that the possession of design experience might have an overall detrimental effect upon a programmers' ability to detect certain types of bugs!

6.3 Experiment 2, program recall

6.3.1 Method

Subjects

Two groups of first year undergraduate students were employed in this study. Participants in both groups had initially at least 6 months experience of Basic

programming. One group of subjects (Group B) attended the same course on program design that was attended by subjects participating in the first experiment. There were 24 subjects in each group. Note that these subjects did not participate in experiment 1.

Materials

The programs used in this study were again based upon a program intended to calculate average rainfall (Johnson and Soloway, 1985) and consisted of 30 lines of Basic code. Following the procedure adopted by Soloway and Ehrlich (1984), two versions of the program were constructed. One version contained five critical lines conveying information relating to three salient plan structures: Calculating a running total, calculating a maximum, and establishing a counter variable plan. Figure 6.4a shows a fragment of this program with two of its constituent plan structures illustrated. A second - unplan-like - version of this program was constructed (see figure 6.4b). In this case, following Soloway and Ehrlich (1984), the initialisation assignments of the count and sum variables violated the normal and correct form of initialisation for the counter variable plan and the running total loop plan. Otherwise the programs were intended to be identical: both version had the same number of lines and a similar number of operands and operators.

Design

The experiment was a three factor design. These factors were:-

1). The type of treatment for each group; exposure to design experience or no exposure (Group A and Group B respectively).

2). The nature of the programs (plan-like or unplan-like).

3). Performance (number of critical lines correctly recalled in a limited time) on first vs second trial.

```
10 REM avrprob
20 LET count = 0
30 LET Sum = 0
40 REPEAT
50 INPUT New
60 IF New = 99999 THEN GOTO 90
70 LET Sum = Sum + New
80 LET Count = Count + 1
90 UNTIL New = 99999
100 IF Count = 0 THEN PRINT "No legal inputs" ELSE PRINT
                              "Average is..... "; Sum/Count
.
.
```

PROGRAM (a) - Plan-like

```
10 REM avrprob
20 LET count = -1
30 LET Sum = -99999
40 REPEAT
50 INPUT New
60 LET Sum = Sum + New
70 LET Count = Count + 1
80 UNTIL New = 99999
90 IF Count = 0 THEN PRINT "No legal inputs" ELSE PRINT
                              "Average is..... "; Sum/Count
.
.
```

PROGRAM (b) - Unplan-like

Figure 6.4. Fragments of programs used in the second experiment. Two of the five critical lines in the program are represented by lines 20 and 30. The plan structures represented (in the case of Program a.) are a Running Total Loop Plan (lines 30 and 70) and Counter Loop Plan (lines 20 and 80).

Procedure

In the first trial subjects were presented with a program on the screen of a microcomputer. Each presentation lasted for 120s. Half of the subjects in each group were presented with the plan-like version of the program and the other half with the unplan-like version. The assignment of programs to subjects was done at random. Immediately after the presentation of the programs subjects were asked to attempt to recall the program verbatim and to retype it onto a familiar full-screen editor. Subjects were given 300s to complete this task. The screen editor was modified so that each depression of the return key (to open a new line) was recorded and time-stamped. This enabled a record to be obtained of the temporal order of recall.

Five months after this first trial a second trial was conducted. This trial followed the same procedure as the first. During the elapsed time between the first and second trials, one group of subjects (Group A) had attended an optional course on program design (the content of this course has been outlined in the introduction to experiment 1). Subjects in both groups attended the same course on Basic programming between the first and second trial. Measures were obtained during each trial of the number of correctly recalled critical and non-critical lines and the order in which these lines were recalled by both groups.

6.3.2 Results

The results of this experiment are shown graphically in figures 6.5 and 6.6. The number of critical lines correctly recalled during the first 100s of the recall session in both trials were entered into a three factor analysis of variance with the factors; Program type (plan-based or unplan-based), Group (A and B) and Trial (first and second). This revealed the following effects;

a). No main effect of Program type, Group or Trial (all F's < 1.5, NS)

b). A three way interactionm between Group, Program type and Trial ($F_{1,5}$ = 14.5, $p < 0.01$).

No other main or interactional effects were apparent.

This three-way interaction can be split up into two separate two-way interactions - one for trial 1, where there should be no significant interaction between Program-type and Group, and one for trial 2 when there should be an interaction. This is indeed the case. For trial 1 the Program type x Group interaction is not significant. For trial 2 this interaction is highly significant ($F_{1,5} = 19.43$, $p < 0.01$).

In order to ensure that both groups exhibited similar levels of programming competence in Basic the results of a simple end of course test were analysed. This test involved presenting students with mini programs that they were expected to debug. The results of this test, which were marked by someone other than the experimenter, were compared for each group and a t-test indicated that no significant differences did in fact exist between the groups.

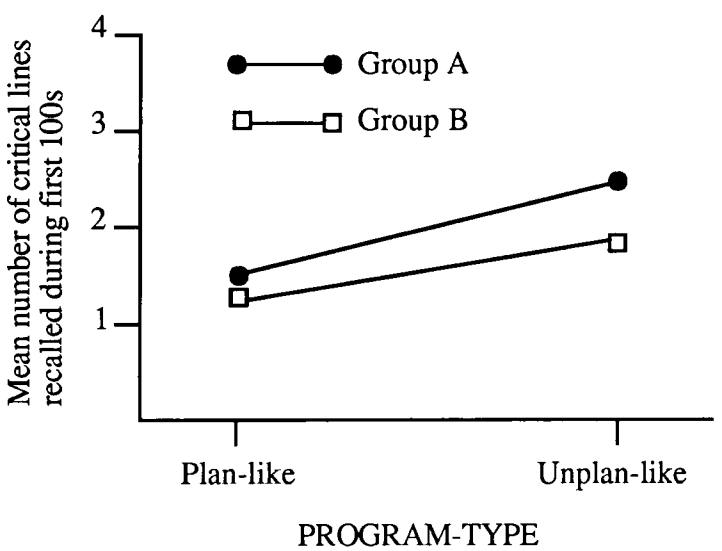


Figure 6.5. Mean number of critical lines recalled (first 100s of recall session) for program-type in trial 1. Group A, design experienced. Group B, non-design experienced.

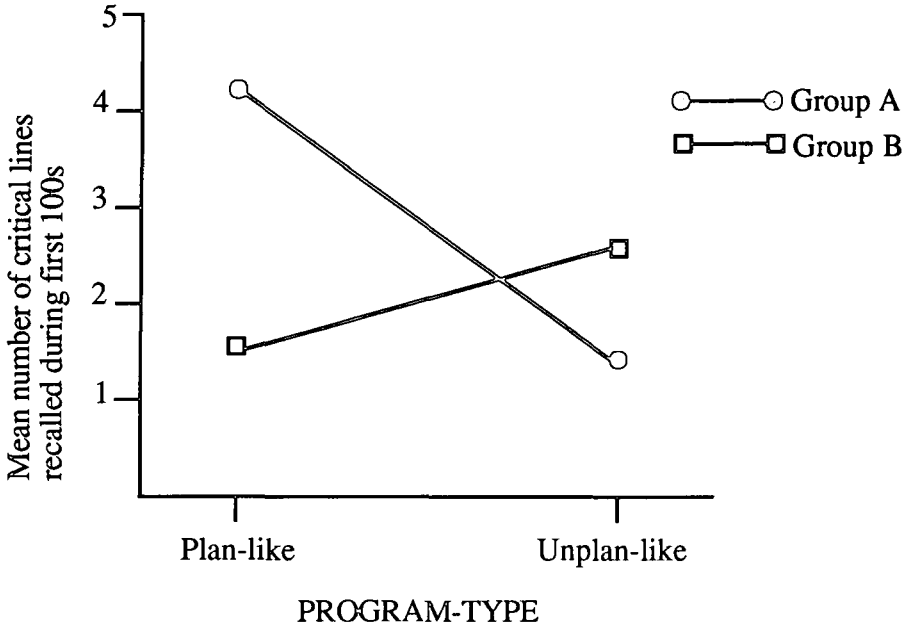


Figure 6.6. Mean number of critical lines reaclled (first 100s of recall session) for program-type in trial 2. Group A, design experienced. Group B, non-design experienced.

6.3.3 Discussion

The only significant effect revealed by the three-way analysis of variance discussed in the results section of this experiment was that existing between Group, Program-type and Trial. This interaction appears to reflect the increased number of plan structures recalled by group A during the second trial. Clearer evidence for this effect is to be found in the results of the individual two-way analyses. These results show that before design training the groups did not differ. However after training, subjects in Group A, the trained group, recalled significantly more plan structures than Group B. The conventional interpretation of the relationship between programming plans and expertise cannot account for this effect. Indeed one might expect plan recall to increase between trials since at the second trial, both groups had attended the same programming course and had

presumably added to their knowledge of the programming language. Indeed, a comparison of the results of the end of course test in Basic for both groups suggests that their expertise (at least in debugging) was broadly equivalent. However contrary to this expectation, if one compares performance by the groups, the increase in the level of recall of plan structures between trails was not consistent.

The results of this experiment suggest that not all experienced programmers use plan-like representations to encode programs. This clearly has implications for those who advocate the programming plan as a key element in their theoretical analysis of expert programming performance. Again the universality of the programming plan appears to be drawn into question.In addition to this, conventional work into programming plans appears to tacitly adhere to the assumption that the recall or the generation of more programming plans is a good way to characterise expertise. However, when proper external measures of performance and expertise are employed, i.e., in terms of debugging abilities, this assumption loses strength.

6.4 Overall discussion

Programming plans appear to form useful constructs only for those programmers who possess design related skills. Indeed, on the basis of the experiment reported above, it would appear that the recall of plan-like structures cannot be used to indicate differences between novice and expert programming performance. This is because one would not expect significant and systematic differences in performance to arise between the two groups of programmers studied. The results of the first experiment reported in this paper confirm that where independent measures are used to assess performance (i.e., in terms of debugging skills) then the difference in overall levels of performance found to exist between those with design experience and those without is not significant. This provides additional support for the conclusions drawn above and, viewed in tandem with these later findings, questions both the universality of programming plans and their use in the characterisation of expert programming performance.

6.5 General conclusions

Taken together the results of the experiments reported in this chapter highlight the effects of design experience upon the nature and development of programming plans. These experiments have shown that the plan theory cannot account straightforwardly for the differences between novice and expert programming performance. In particular, they clearly show the important role played by the acquisition of design-based knowledge in the comprehension of programs. The conflict between the natural and the notational approach to programming plans may, when viewed in the context of these experiments, turn out to be more apparent than real. However, the main conclusion remains valid. That is that current views concerning the nature and development of programming plans are flawed in two ways. On the one hand, the notational view is too narrow in its perspective because of the emphasis it places on notation at the expense of other demonstrably important factors. On the other hand, the views expounded by the Soloway group reflect a fundamental confusion between the measurement of plans and their use in theoretical explanations of expert performance. Hence, neither provides a sound theoretical basis for a full psychological theory of programming.

Clearly the naturalistic and notational views of programming plans are by no means mutually exclusive. One might wish to suggest that 'natural' plans exist, but that novices have difficulty expressing them in a programming language. However, some programming languages may reveal plan components more clearly than others (the notational view). A tripartite analysis is proposed here in which both of the views described above can be considered valid only when the role of design experience is recognised as an important factor in the development of plan-related knowledge. By adopting this view, it is possible to provide a consistent analysis of existing experimental data.

Using this framework, it can be argued that while novices have some difficulty learning to express plans, they can benefit from training in design. This is because although training in design is not concerned with the explicit teaching of programming plans, we can see that the design process provides a means of applying the salient features of plans, and discovering the links between them.

Thus such training is likely to aid the programmer in constructing and/or employing the vital mapping between structures in the problem domain and structures in the language domain.

It appears from the experiments reported in this chapter that design training may encourage programmers to focus upon plan structures during comprehension and debugging activities. A similar finding has been reported by Stone, Jordan and Wright (1990), who demonstrated that training in structured programming techniques, such as those reported in this chapter, can improve debugging performance by "increasing the comprehension of program goals and plans" (pg 81). They go on to claim that their results "suggest that the value of structured programming techniques may be realized more in the programmer's way of thinking about a program than in the creation of a structured program per se." (p. 81).

Subsequent experiments reported in this thesis suggest that the development of expertise in programming does not simply involve the accumulation of plans. Rather, programming expertise appears to depend upon the structuring of programming knowledge such that certain salient plan elements can be retrieved and accessed more quickly. It may be the case that design training facilitates this structuring process by encouraging programmers to focus upon the salient elements of plans. In addition, this might be expected to enhance the mapping between the language and problem domain that is discussed above, by providing a means of applying the salient features of plans and establishing the links between them.

The model of programming expertise presented in this thesis will be considered in greater depth in chapter 12, where the results of this and subsequent experiments are interpreted in a rather more integrated and global context. The experiments reported in this chapter contribute to this model, but must be viewed in tandem with the other experiments reported in this thesis. Nevertheless, the experiments reported in this chapter raise a number of specific issues for the plan theory of programming. Firstly, despite a suggestion to the contrary in earlier work, programming plans do appear to be generalisable to languages other than Pascal. However, only the design experienced programmers involved in this experiment

demonstrated significant plan use. Secondly, there does not appear to be a clear relationship between the possession of plans and the development of expertise in programming. Hence, programmers who might be regarded to possess equivalent levels of skill, in terms of their general debugging ability, do not necessarily use plans to the same extent. These two findings appear to challenge the central tenets of the plan theory as it is currently expressed and clearly raise fundamental doubts about its ability to account for the nature and the development of programming expertise.

Chapter 7. Plan Violation and Programming Expertise: Evidence for knowledge restructuring

7.1 Introduction

One of the main conclusions of the previous experiment was that programming plans do not appear to provide a straightforward account of programming expertise. Hence, programmers may exhibit the same general level of programming competence in the context of certain tasks, but use plans rather differently. One area that has yet to be subjected to experimental analysis is concerned with the dynamic aspects of plan use and the relationship between plan use and the development of expertise.

One major criticism of the plan/goal analysis of programming is that it presents a fairly limited view of the programming activity. This is particularly true given what we know about the role of plans and goals in other problem solving domains. Programming plans are proposed as constructs that form the basis for distinctions between novice and expert performance, yet little concern has been directed toward an analysis of the development and use of plan structures and the refinement of goals as programming expertise and knowledge increases.

Studies of the development of plan structures and goals in other domains suggest that the plans that underpin expertise develop through a number of identifiable stages. Kay and Black (1984, 1986), for example, have traced the plan acquisition process in a text editing domain. They suggest that a complex relationship exists between the development of plan structures and increase in expertise. They highlight the importance of the refinement of plan structures and the use of selection rules as expertise develops.

In light of this work, the notion that the primary distinction between the novice and expert programmer is solely based upon the latter's possession of plan related structures would appear to be an oversimplification. Indeed, experts need to not only possess plan structures but also know how to use them appropriately. Kay and Black (op cit) suggest that during intermediate stages of skill acquisition plan structures are already well developed but that genuine expertise tends to be

exhibited only when appropriate selection rules are developed to guide the implementation of plans.

Kay and Black suggest that skill learning in the text editing domain progresses through four identifiable stages. The first stage in this description represents the naive user who has no text editing experience. They suggest that users bring to the task a range of preconceptions about text editing terminology which may or may not accord with their interpretations of that terminology as expertise develops.

The next stage in Kay and Black's description is concerned with the goal of overcoming this prior knowledge bias. They suggest that during this phase (which they call the initial learning phase) users develop conceptual knowledge structures that link specific goals with commands. At this stage, users tend to cluster together functionally related commands. For example, INSERT, PUT, REPLACE might be grouped together because these commands are used to accomplish the goal of adding information. Users tend to modify their initial clustering strategy, based upon prior knowledge associations, to one which emphasises the functional links between commands.

The third phase of expertise development is concerned primarily with the formation of plans. Once users have acquired a range of basic editing commands and goals, they learn that a number of commands can be grouped together in terms of the frequency of the use of such commands in accomplishing a particular goal. That is, they combine the actions that were organised separately during the phase of initial learning. Both Kay and Black and Sebrechts et. al. (1985) provide evidence about the nature of the development of knowledge structures during this third phase. During early stages users group commands in terms of their functional relationships. As expertise develops this grouping tends to occur with respect to commands that are used in conjunction to accomplish a particular goal. For example, the commands PUT and PICK might be grouped since they are used together when the user wants to move an item of text.

During the final stage in Kay and Black's model, users produce compound plans to accomplish major goals and refine the selection rules that are used to choose among alternative plans in given situations. At this level goals are linked to plans using the conditions in which these compound plans are invoked, whereas, in

phase three goals are linked to simple plans, and during phase two merely to actions.

The Kay and Black model suggests that a complex relationship exists between expertise and the development of goals and plans in a routine cognitive activity such as text editing. This model also has a number of implications for our understanding of the role of goals and plans in the programming domain. Previous studies of the programming activity suggest that expertise can be characterised primarily by the possession of plans or plan related structures and additionally that the existence of such plans can be used to make the distinction between the novice and expert. In the programming domain little or no concern has been directed toward an analysis of the development and refinement of plan structures as expertise increases. The Kay and Black description of skill development suggests a number of key areas of concern for the analysis of problem solving in programming. These can be summarised as follows;

i). Can the mere existence of plans be taken as an indicator of expertise?

ii). Are plan structures exhibited at intermediate skill levels?

iii). Do programmers develop plan selection rules as their expertise develops?

iv). How are plan structures refined as expertise develops?

These issues are here addressed via an experimental study of the programming activity as programming skill increases. Programmers of varying skill levels (Novice, Intermediate and Expert) were presented with a number of Pascal programs, each of which contained several blank lines. In addition, a number of program fragments were presented with each program. The programmer's task was to attempt to state which of the fragments could be used to best complete the program; which might be their second choice, and so on. In the first series of programs the associated program fragments represented plan structures and contraventions of plan structures.

For example, Figure 7.1 represents a program intended to calculate the square root of a number. The important plan structure in this program might be referred to as a Data Guard Plan (Soloway and Ehrlich, 1984). The first program fragment (1) illustrates the correct use of the Data Guard Plan. This plan protects the "Sqrt" function from trying to take the square root of a negative number. The 'IF' part of the statement carries out this check and makes the number positive if necessary.

The second program fragment (2) represents a contravention of the Data Guard Plan. Here the first statement suggests an assignment type initialisation (Num = 0). This gives rise to the expectation of an assignment update (i.e., Num := Num +1). However, the Data Guard Plan predicts a read update since using an assignment statement would never result in a negative number - making the Data Guard Plan in this case superfluous. The third program fragment (3) contravenes the plan structure in a more straightforward manner and simply introduces an incorrect test for a negative value (Num > 0) in the last statement.

One might expect, in light of the studies into the development of expertise that have been reviewed in this chapter, that the level of expertise possessed by the programmer would have some effect upon both their choice of the ordering of program fragment and the time taken to make this choice. If the existence of plan structures can provide an indication of the programmers expertise or, similarly, if such structures facilitate the comprehension of programs, then expert programmers might be expected to choose a fragment that best completes the program which represents or conforms to a plan structure.

Indeed, from the results of the Kay and Black studies reviewed in this chapter, it might also be expected that programmers of intermediate skill level would make a similar choice. This would confirm the finding that plans exist and are used at both intermediate and expert skill levels. If plan structures are well represented as cognitive schemata, which would be expected in the case of the expert, then the time needed by the programmer to make a choice of appropriate program fragment (conforming to a plan structure) would presumably be less than that needed when such structures are poorly formed (as one might expect in the case of the novice) or when plan structures remain unconsolidated (corresponding to the intermediate level).

The second series of program fragments used in this study represented the correct and violated use of program discourse rules. Program discourse rules, according to Soloway and Ehrlich, specify the conventions in programming. For example, such rules might take the form (following Soloway and Ehrlich, 1984);

"If there is a test for a condition then the condition must have the potential of being true."

"An IF should be used when a statement body is guaranteed to be executed only once and a WHILE used when a statement body may need to be executed repeatedly"

To a large extent rules of programming discourse correspond to the types of selection rules used by experts that have been identified in other studies of skilled performance. Soloway and Ehrlich (1984), for example, claim that discourse rules govern the use of plan structures in programming. That is, they provide an indication of the type of plan that is appropriate at a given point in the program and, in addition, provide some constraints on the formation of plan structures. Figure 7.2 illustrates the representation of a discourse rule and two violations of the rule in three program fragments. The discourse rule represented in the program is *that variable names should reflect their functions*. In this example the program is intended to calculate either a maximum or a minimum value.

The first program fragment (1) represents a procedure that conforms to the program discourse rule. That is, the variable name Max is used in a procedure intended to calculate a maximum value. The second fragment of code (2) represents a violation of this discourse rule. This procedure uses the Max variable in conjunction with the calculation of a minimum value. The last procedure (3) represents a similar violation of this discourse rule but in this case the program would produce a run-time error if executed.

Again, we might expect experts to exhibit a preference for the program fragment that represents the correct use of a discourse rule, since at this level of skill the selection rules corresponding to the rules of programming discourse will be well

developed. Kay and Black suggest that expertise is characterised not only by the possession of plans but also by the use of appropriate selection rules. This also conforms to predictions that stem from the GOMS model (Card, Moran and Newell, 1980) of skilled behaviour which places an emphasis on the presence of selection rules as a characteristic of expertise.

At intermediate skill levels it might be expected that programmers, while displaying a preference for plan structures, may not exhibit well developed selection rules. Hence one might expect that the choice of program fragment used in the completion of a program may not accord with a strategy based upon the use of such rules. Correspondingly, no particular preference for the program fragments that represent discourse rules should be exhibited. This effect should also be apparent in the case of the novice programmer. The time taken to produce an appropriate ordering of program fragments should also provide some evidence about the development and role of program discourse rules at different skill levels. In the case of the expert programmer one might expect discourse rules to be well developed. Hence, the time taken to decide upon an appropriate ordering of rules in a particular circumstance would be less than that required by both the novice and the intermediate programmer. In such cases we would hypothesise that rules of programming discourse remain undeveloped.

7.2. Method

Subjects

A total of 45 subjects participated in the experiment. These subjects were categorised according to experience into Novice, Intermediate and Expert programmers. Each of these groups were of equal number. The Novice programmers were undergraduates with approximately two months experience of Pascal. The Intermediate group were also undergraduates but this group had completed a nine month course in Pascal. The Expert group were either teachers of Pascal or were employed in industry as programmers. All in the latter group had used the language on a regular basis for more than two years.

Procedure

Subjects were given the programs and program fragments (See Figure 7.1 and Figure 7.2) in booklet form. These booklets also contained instructions for the completion of the task. Subjects were asked to attempt to complete all the programs (i.e., to choose the most appropriate ordering of program fragments) as quickly as possible. No time limit was imposed on this task and all subjects responded to all the programs. Both the ordering of program fragments and the time taken for each subject to decide upon this ordering for each program was recorded.

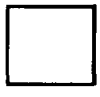
```
PROGRAM One (input, output),
  VAR Num REAL,
      I   INTEGER,
  BEGIN



    Writeln (Num, Sqrt (Num)),
   END,
 END
```

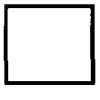
(1)

```
FOR I = 1 TO 10 DO
   BEGIN
      READ (Num),
      IF Num < 0 THEN Num = -Num,
```



(2)

```
Num = 0,
FOR I = 1 TO 10 DO
   BEGIN
      READ (Num),
      IF Num < 0 THEN Num = -Num,
```



(3)

```
FOR I = 1 TO 10 DO
   BEGIN
      READ (Num),
      IF Num > 0 Then Num = -Num,
```

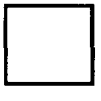


Figure 7.1. A program intended to calculate a Square root illustrating program fragments corresponding to (1) the correct use of the Data Guard Plan and violations of its use (2) and (3).

```
PROGRAM Three (input, output),
  VAR Max, I, Num, INTEGER,
   BEGIN

     END,
  Writeln (Max),
  END
```

(1)

```
Max = 0
FOR I = 1 TO 10 DO
 BEGIN
   READLN (Num),
   IF Num > Max THEN Max = Num
```

(2)

```
Max = 999999
FOR I = 1 TO 10 DO
 BEGIN
   READLN (Num),
   IF Num < Max THEN Max = Num
```

(3)

```
Max = 0
FOR I = 1 TO 10 DO
 BEGIN
   READLN (Num),
   IF Num = Max THEN Max = Num
```

Figure 7.2. A program illustrating program fragments representing the correct use of a program discourse rule (1) and violations of that rule (2) and (3).

Materials

Two sets of programs and program fragments were used in this study. The first set of three programs were associated with program fragments that represented plan structures and contraventions of plan structures. The three program types were as follows:

Program Type1 - A program intended to calculate a square root (See Figure 7.1).

Program Type 2 - A program intended to calculate an average.

Program Type 3 - A program intended to calculate a maximum or a minimum value. (See Figure 7.2).

The plan structures employed (derived from Soloway and Ehrlich, 1984) were as follows:

Plan 1 - A Guard Plan (see Figure 7.2).

Plan 2 - A Running Total Loop Plan.

Plan 3 - A Search Plan.

Plan 1 was associated with Program 1, Plan 2 with Program 2 and Plan 3 with Program 3. Each program was presented with three program fragments (as in Figures 7.1 and 7.2). One program fragment represented the correct use of the plan and the second and third fragments a contravention of plan structures. The ordering of program fragments presented with each program was randomised, as was the order in which program types occurred.

The second series of programs consisted of the same program types (Program Type 1-3) and a number of associated program fragments. These fragments represented program discourse rules and violations of these rules (See Figure 7.2). The three discourse rules used (Derived from Soloway and Ehrlich, 1984) were as follows:

Discourse Rule 1 - "If testing for a condition then the condition must have the potential of being true."

Discourse Rule 2 - "Do not include statements in the program which will not be used."

Discourse Rule 3 - "Variable names should reflect function."

Each of the three program types was presented with three associated program fragments. One of these program fragments represented the correct use of the program discourse rule, a second represented a violation of the rule, but resulted in an executable program, and the third violated the discourse rule but resulted in a error-prone program (See Figure 7.2). In the same manner as above, discourse rule 1 was associated with program 1, discourse rule 2 with program 2 and discourse rule 3 with program 3.

7.3. Results

Figures 7.3 to 7.6 illustrate the results of this study. Figure 7.3 shows the programmer's first choice of program fragment when completing a program. These fragments correspond to either a plan structure or to plan structure violations. As can be seen, both intermediate and expert programmers choose, in the main, to complete the program with the fragment that corresponds to the correct use of a plan structure. Conversely, novices do not exhibit a preference for plan structures over program fragments representing violations of plan structure. These effects were statistically significant.

There was an overall efect of skill level ($F_{2,28} = 13.64, P < 0.001$) and of fragment type (plan/plan violation) ($F_{2,28} = 10.62, p < 0.001$). In addition, the interaction between skill level and fragment type (plan/plan violation) was also significant ($F_{4,56} = 5.92, p < 0.001$). Multiple post-hoc comparisons were conducted with the Newman-Keuls test, and a significance level of $p < 0.01$ was adopted for all such tests.

This procedure indicated that both expert and intermediate programmers choose the program fragments representing plan structures more frequently than novices. Correspondingly, novices tended to choose fragments representing plan structure violations more frequently than both intermediate and expert programmers. None of the other contrasts between means was significant. Hence, novices choose fragments representing plan structures with approximately the same frequency as they choose fragments representing plan structure violations and intermediates choose to use the correct plan fragment with about the same frequency as experts.

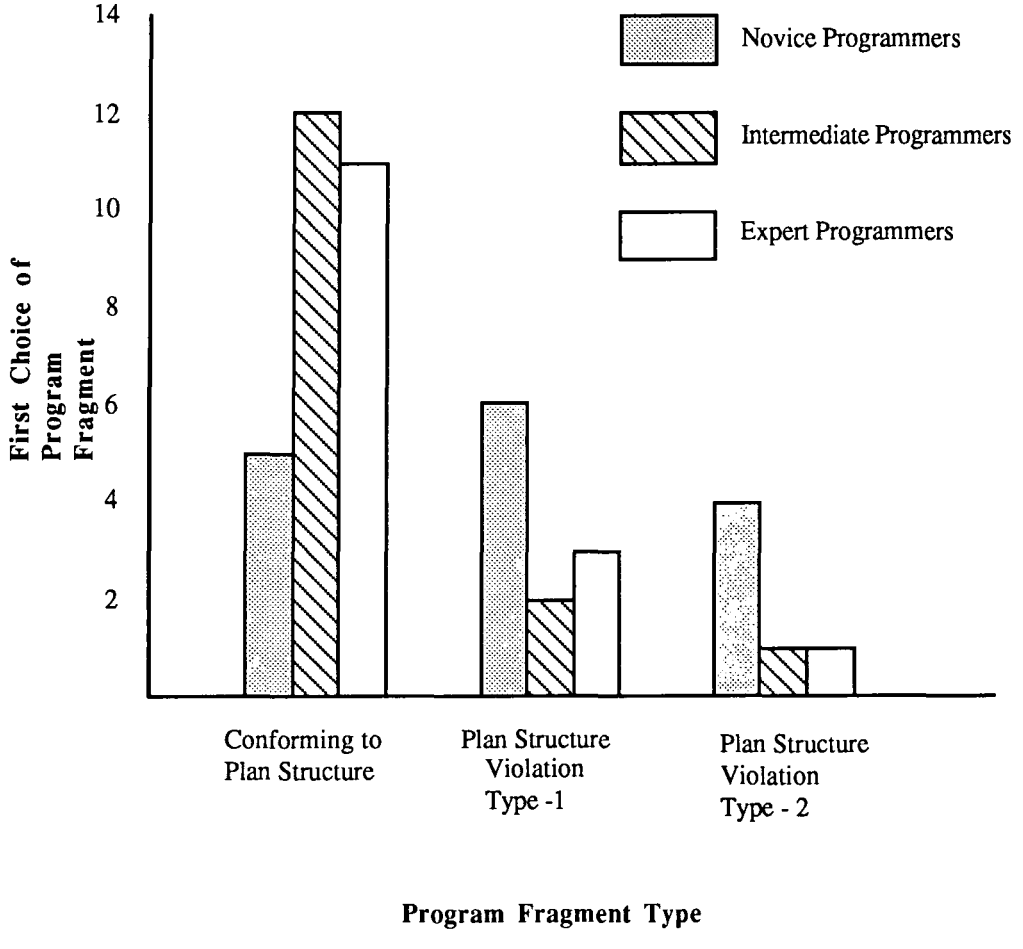


Figure 7.3. The frequency with which a program fragment (corresponding to a plan structure and violations of plan structure) was choosen as best completing a program by novice, intermediate and expert programmers.

Figure 7.4 shows the total time taken by novice, intermediate and expert groups to make a choice of program fragments and to order them appropriately. Novices take the greatest time to order the program fragments, follwed by the intermediate group and then by experts. These differences are statistically significant. Novices take significantly longer than those in the intermediate group to order program fragments (t-test, $p < 0.05$, two-tailed), and the latter group also take somewhat longer than experts (t-test, $p < 0.05$, two-tailed).

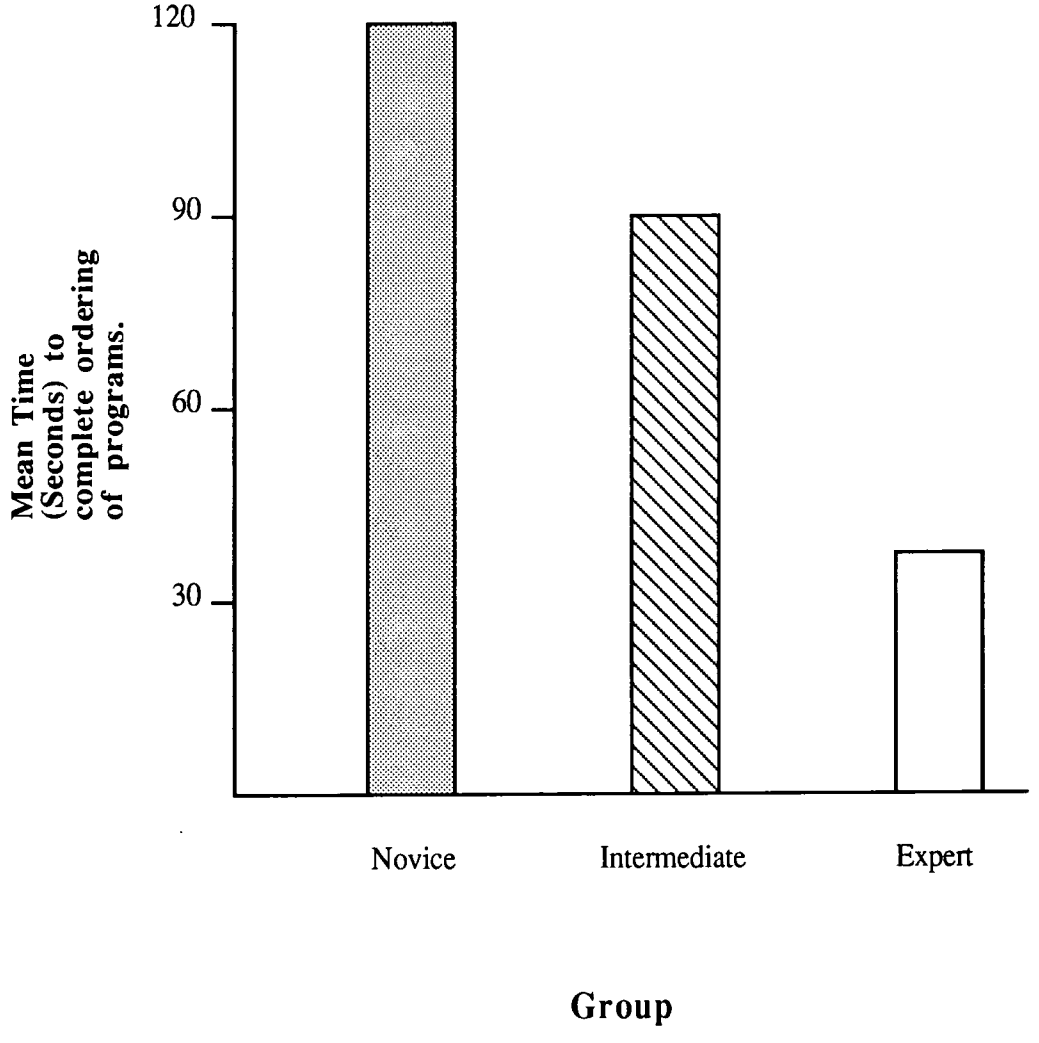


Figure 7.4 . Mean time taken to order program fragments (corresponding to a plan structure and to violations of plan structure) by novice, intermediate and expert programmers.

Figure 7.5 shows the mean number of occasions in which program fragments, corresponding to the correct and violated use of a program discourse rule, are chosen first in the ordering of program fragments by novice, intermediate and expert programmers. Here, when asked to complete a program, the expert group tend to choose first the program fragment that represents the correct use of a program discourse rule. Those in the intermediate group tend to make this choice less frequently. Novices exhibit no particular preference for the correct use of a discourse rule over program fragments representing contraventions of the rule. These effects were statistically significant.

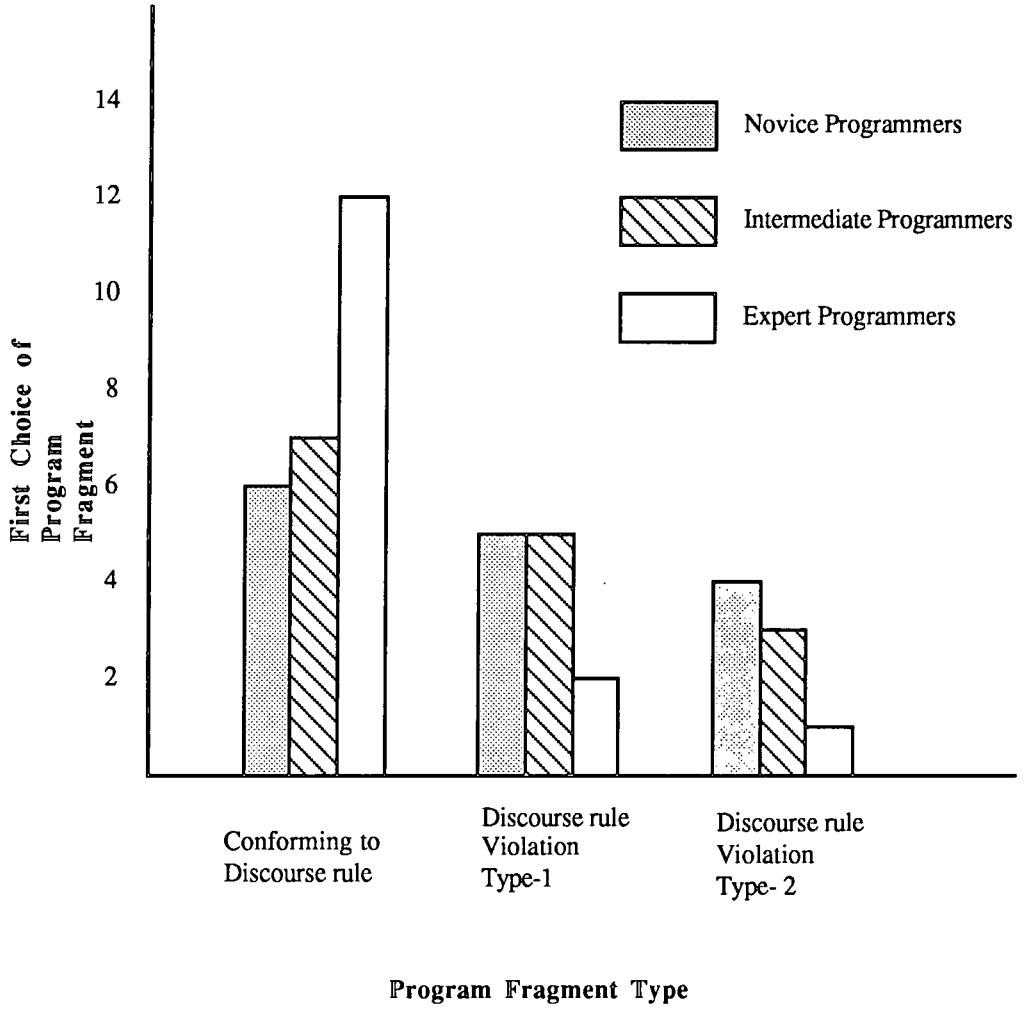


Figure 7.5. The frequency with which a program fragment (corresponding to a discourse rule and to violations of a discourse rule) was choosen as best completing a program by novice, intermediate and expert programmers.

There was an overall effect of skill level ($F_{2,28} = 8.63$, $p < 0.001$) and of fragment type (discourse rule/discourse rule violation)($F_{2,28} = 19.54$, $p < 0.001$). The interaction between skill level and fragment type (discourse rule/discourse rule violation) was also significant ($F_{4,56} = 11.46$, $p < 0.001$). Once again, multiple post-hoc comparisons were conducted using the Newman-Keuls test, and a significance level of $p < 0.01$ was adopted. This procedure indicated that experts choose the program fragment representing the correct use of a program discourse rule more frequently than both the novice group and the intermediate group. Comparing the choice of program fragments representing the correct and violated use of a program discourse rule suggests that experts tend to choose the program fragment that represents the correct use of such a rule more frequently than a fragment representing a contravention of the rule. The same is true of the intermediate group. Conversely, novices show no preference for fragments representing the correct use of a discourse rule compared with those representing the controvention of such a rule.

Figure 7.6 shows the total time taken by novice, intermediate and expert groups in completing the ordering of program fragments representing discourse rules and discourse rule violations. Here, experts tend to complete the task faster than both intermediate and novice groups. This is confirmed by further statistical analysis. Experts order program fragments faster than both those in the intermediate group (t-test, $p < 0.05$, two-tailed), and those in the novice group (t-test, $p < 0.05$, two-tailed). The intermediate group, in turn, appear to complete this ordering task slightly more quickly than those in the novice group, but this difference was not significant.

7.4. Discussion

The results of this study suggest that the relationship between programming skills and the formation and utilisation of plans and selection rules in programming is by no means straightforward. Previous work in this area suggests that expertise is characterised primarily by the possession of programming plans and rules of programming discourse (Soloway and Ehrlich , 1984).

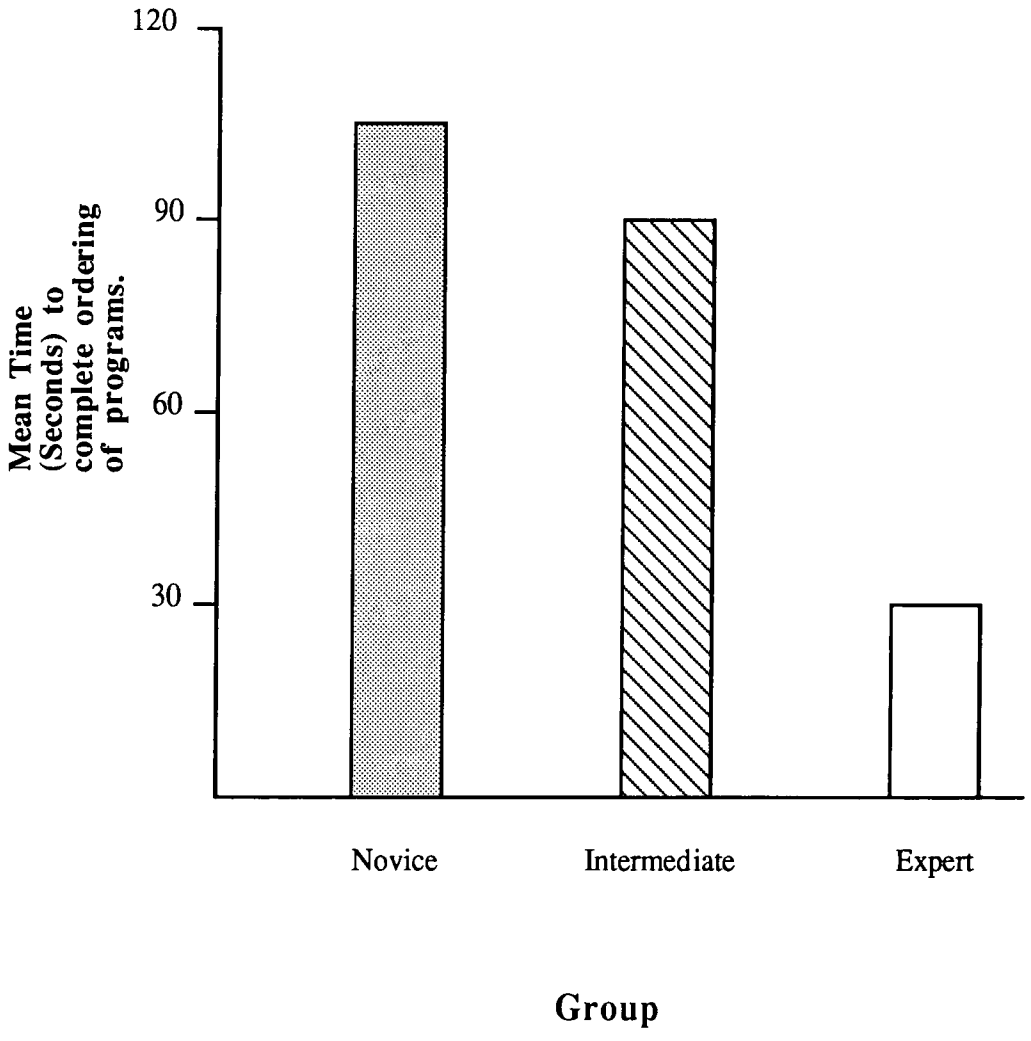


Figure 7.6. Mean time taken to order program fragments (corresponding to a discourse rule and to violations of a discourse rule) by novice, intermediate and expert programmers.

This work, however fails to examine the way in which such plans might be formed and the nature of their use as skill in programming develops. The results of the experiment reported above suggest that programming plans exist at both expert and intermediate skill levels in programming. At both levels such plans provide a basis for the comprehension of programs. Both expert and intermediate groups are equally likely to choose a program fragment corresponding to a plan structure when completing a program. Novices, in contrast appear to exhibit no

particular preference for those program fragments representing plan structures over those representing plan structure violations.

One interesting difference in performance that emerged between the expert and intermediate groups was that reflected in the time taken to complete the ordering of program fragments corresponding to plan structures and to plan structure violations. The intermediate group took much longer than the expert group to produce such an ordering. This may suggest either a). that while both intermediate and expert groups utilise plan based structures, in the case of the former these plans remain unconsolidated or b). that the intermediate group are, for some reason, unable to easily access or activate these plans.

These findings would accord with results obtained from studies of the development of skilled performance in other problem solving domains. For example, Kay and Black (1984) suggest that the existence of plan structures in text editing is characteristic of both intermediate and expert performance, but that the rules of selection governing the use of appropriate plan structures only develop at higher levels of expertise. Such expertise would appear, in addition, to be characterised by the development of so called compound plans. Indirect evidence for this is adduced by Kay and Black from studies which suggest that as expertise develops the time taken to formulate and implement plans decreases. Another possible interpretation of this result might be to suggest that experts automate some simple generic sub-components of the programming task (these may correspond to plan structures). Empirical studies of this knowledge compilation process have been reported for general problem solving tasks (Anderson, 1982) and within the more specific context of programming (Anderson, 1987; Wiedenbeck, 1985).

The results of the experiment reported in this chapter provide some support for the contention that expertise in programming cannot be explained merely by alluding to the notion of the so called programming plan. Such plans are used by both intermediate and expert programmers. The important distinction between these groups would appear to be based more strongly upon the use and deployment of appropriate selection rules as expertise develops.

This distinction appears to be reflected in the results stemming from an examination of the role of program discourse rules in skill development in programming. Expert programmers tend to make the most use of program discourse rules. Their use of such rules differs significantly from that of intermediate programmers who, when completing a program, tend to exhibit no particular preference for the program fragment representing the correct use of a discourse rule over those violating such rules. This would suggest that an important characteristic of expertise is related to both the possession and use of program discourse rules. Examining the time taken to order program fragments which correspond to discourse rules and to violations of these rules indicates that intermediate performance differs little from that of novices. Experts, on the other hand, perform the task a great deal faster than both novice and intermediate groups. This again supports the view that such discourse rules are an important feature of expert performance.

Existing plan/goal analyses of programmer behaviour provide only a limited insight into some of the underlying features of this important problem solving activity. Studies in other domains, most notably that of text editing, suggest the need to examine in more detail not only the role of plans but also the nature of the development and refinement of such plans as expertise increases and, in addition, the central role played by selection rules in expert performance. This chapter has attempted to highlight the correspondences that exist between plan/goal analyses of text editing and those which appertain in programming. Strong similarities have emerged between these domains.

For example, models of problem solving such as that proposed by Kay and Black for text editing provide a valuable basis for an analysis of programming. Extending the scope of such models to account for performance differences in programming has highlighted a number of difficencies in the current plan/goal analysis of problem solving within this domain. The present study has attempted to address some of these difficencies and by doing so to suggest ways in which the plan/goal analysis of programming might be extended. The central theme of the chapter - that programming plans alone do not provide an adequate basis for a full account of expert problem solving in programming - is supported and the plan/goal analysis of programming is extended to reflect the central role played by selection rules in expert performance.

7.5 General Conclusions

The results of the experiment reported in this chapter provide general support for the idea that the possession of plans per se does not necessarily guarantee the same level of programming performance for different groups of subjects. It has been suggested that performance differences may be associated with the development and refinement of appropriate selection rules and/or with a process of knowledge restructuring that may result in the development of compound plans.

In addition, this process may give rise to the restructuring of knowledge *within* plans and lead to certain structures becoming prominent within the context of individual plan structures. The model of programming knowledge presented later in this thesis suggests that via this restructuring process, focal plan elements will tend become more accessible as expertise develops. Hence, in the context of the present experiment, while intermediates appear to be able access the same plan knowledge as experts, they are able to achieve this more effectively, as evidenced by their greater speed. It may be suggested that this increase in speed results from the greater ease with which plans can be retrieved and implemented when focal structures are accessible.

level attained by the programmer. This chapter endeavours to extend current models of programming by emphasising the need to consider in detail not only the development of programming strategy, but also the way in which knowledge representation and features of the task language interact to give rise to particular forms of programming behaviour.

The studies described previously in this thesis suggest a particular dichotomy between theories of programming that emphasise knowledge representation and those which stress the effects of language notation. However, these two elements are by no means mutually exclusive. The problem facing such theories is one of providing explanations for the way in which these factors interact to produce observed phenomena.

8.1.2 Strategy vs Knowledge

Also, besides questioning the assumed universality of the programming plan, the work cited above suggests other problems with the plan theory of programming. Bellamy and Gilmore (1990), have compared the coding behaviour of experienced programmers using a number of different languages. The intention of this work was to examine whether the order of program generation suggested the existence of plan-like structures. Their evidence for the use of plan structures in program generation was equivocal (see chapter 3).

Hence, the question that arises is why plans should prevail in the comprehension process (as in Soloway's recall experiments) but not during generation? It may be that the appearance of plan-based behaviour is determined by comprehension strategy rather than knowledge (see chapter 4). Hence, studies which have examined recall as opposed to generation may, as Bellamy and Gilmore suggest, have tapped post-hoc rationalisations of the programmer's behaviour. Therefore, it would seem reasonable that studies investigating such behaviour should aim to clarify or make explicit the particular role of strategy versus knowledge. In addition, it is clear that studies examining program recall should not be assumed necessarily to be tapping the same knowledge structures or programming strategies as those found to exist in code generation.

8.1.3 Addressing Strategy

Other problems with the plan notion are that little attention has been paid to a consideration of the mechanisms that control plan selection or implementation, to the nature of the development of plans with expertise or to the dynamic aspects of plan use (see chapter 3). Rist's (1986, 1989) model of the programming activity suggests that novices and experts employ very different strategies when developing a program to solve a particular problem. Rist claims that expertise is characterised by the ability of the programmer to focus upon the most salient parts of the plans which comprise a program. These Rist terms the 'focal lines' of the plans. Rist suggests that as expertise develops, some plans are automated (such as input and output) and initially ignored during design. This enables the programmer to direct her attention to the more difficult or novel segments of code. In terms of Rist's framework, as expertise increases plans are selected rather than constructed, and knowledge of the plan focus reflects this increase in expertise.

In contrast to Rist's model, Green, Bellamy and Parker (1987) suggest that when code is not generated in a strict linear fashion (which they claim is the 'natural' development path for the construction of programs; cf Hoc, 1981), this is primarily because of problems with notational features of the programming language (partly because of it's limited 'role-expressiveness') or because of constraints imposed by the device language. Besides these two determinants of strategy, Green et al (1987), also suggest the importance of the programmer's knowledge representation, but in contrast to previous studies that have emphasised the role of knowledge representation, their empirical work has focused upon an investigation of those features of both the task (programming) language and the device language which are thought to determine strategy.

8.1.4 Towards an integrated developmental framework for understanding programming behaviour.

It is likely that features of the device language, the task language and the programmer's knowledge representation interact to determine the nature of programming strategy. The work reported in this chapter provides empirical

support for a model that has been constructed in order to explain the development of programming strategy and to clarify the nature of the interactions between programming language and the development of those structures which are hypothesised to represent programming knowledge. The aim of this work is twofold. Firstly, to provide additional support for the models of coding presented by Green et al and by Rist and secondly, to extend and elaborate these models by exploring the way in which their separate aspects (i.e., Green, Bellamy and Parker's emphasis on task language features and the programming environment and Rist's on knowledge representation) might be combined to form a single and unified developmental framework. This framework aims to show how programming strategies change with changes in knowledge representation arising via restructuring and to highlight the effects of the notational features of the task language on the development of these strategies.

In order to investigate the strategic aspects of the programming activity the work reported here used a method similar to one originally devised by Green et al to examine non linearities in the coding process. Green et al, used discontinuities or 'jumps' in the generation of program text to indicate departures from linearity. For this purpose a jump was defined as an editing action which was followed by moving the cursor to another location and performing another editing action. An extension to this method, which taps more directly the role of knowledge representation in the determination of programming strategy, involves examining departures from linearity within and between the program's plan structures. This method has been used successfully to investigate the more general effects of programming language notation and skill differences on strategy (Bellamy and Gilmore, 1990).

Essentially, the method involves identifying the plan structures in code that have been generated or reconstructed from memory and then analysing, from transcripts of the coding activity, the number of jumps made between lines within the same plan structure (intra-plan jumps) and the number of jumps made between lines which form part of different plan structures (inter-plan jumps).

Such jumps can be characterised as the points at which programmers make some change in their code. Gray and Anderson (1987) introduce the notion of

'change-episodes' to describe these key junctures in the coding process. They suggest that change-episodes can be implemented in two distinct ways; either as minor local amendments to a program or as major transformations in the programs' goal structure.

Inter and Intra-plan jumps loosely correspond to the categorisation of change-episodes proposed by Gray and Anderson. For instance, Intra-plan jumps involve small local changes to code while inter-plan jumps may (though will not always) imply some change to the program's plan or goal structure. The intention of the work reported in this chapter is to extend out current understanding of the development of programmers' knowledge representations with increasing expertise and to investigate the more general effects of the notation of particular languages within this broadly developmental framework. Using the technique outlined above a number of issues might be addressed.

Firstly, if plan structures constitute the underlying cognitive representation of a program and are not language dependent, but are instead related to the programmer's level of expertise, then clearly this will be reflected in differences in the strategic use of plan structures by programmers of different skill levels.

Secondly, this technique provides a means of assessing the way in which the effects of language notation might facilitate or discourage plan use. In contrast to the method employed by Green et al, which simply analysed the number of distinct non-linearities in coding, here these non-linearities can be examined within the context of discrete plan-based knowledge structures. This provides a means of examining the interactions that might exist between features of the notation of the programming language and the programmer's knowledge representation. It may also be the case that features of the notation of programming languages tend to assume a greater or lesser role in the determination of strategy as programming skill develops. Hence, this technique enables us to examine the more complex interactions that might exist between expertise, knowledge representation and notation.

A second measure that provides information about the factors that affect or contribute to programming strategy is pause data. Such data has been used to

indicate the independence of discrete plan structures in memory. For instance, Haberlandt (1980) has found reading time evidence for story episodes as independent memory units. Specifically, Haberlandt found that readers paused for a greater length of time at the beginnings and ends of episodes in stories. In a similar way, Robertson and Black (1983) have shown that pause time increases between hypothesised plan boundaries in a text editing task. Reitman and Rueter (1980) have invesigated the organisation of programmer's knowledge represenations using a free recall technique backed up with collateral converging evedence obtained from structures induced from the pattern of recall pauses (see chapter 3).

Within the present context we are interested in the time spent pausing between the execution of inter- and intra-plan jumps. This information will have a twofold use. Firstly, it will provide evidence for plan boundaries; hence the pause time between intra (within) -plan jumps should be less than that occurring between inter (between) -plan jumps. Secondly, such data will allow us to investigate issues such as whether the ability to locate plan boundaries may differ as a function of expertise or is dependent upon salient features of the programming language notation. Indeed, the interaction between these elements may turn out to be more revealing. For example, it may be the case that the discriminability of language structures, which in turn is dependent upon notational features such as 'role-expressivness', may have a significant role to play as programming skill develops, but becomes less important at higher levels of expertise. In this case, the ease or difficulty of discriminating between plan structures will be reflected in the time spent pausing between inter-plan jumps.

8.2 Method

Subjects

Thirty Six subjects were recruited for this experiment. These subjects were classified into 3 groups of equal size according to their programming expertise. The novice group consisted of first year undergraduate computer science students all of whom had attended a preliminary short course in Pascal. All subjects in this

group had some knowledge of Basic, although this was limited to experience acquired during ad hoc courses prior to their matriculation. None of these subjects expressed the feeling that they could claim any particular expertise in Basic. Indeed this prior screening of potential subjects excluded a number of subjects from this group because of their wide ranging experience of Basic and concomitant knowledge of the language.

A second group of subjects was classified as intermediate. This group consisted of second and final year computer science undergraduates. Subjects in this group had completed two single term courses in Pascal, and all had employed the language extensively in project work. All subjects in this group professed to being reasonably conversant with Basic. Indeed, most of the subjects classified as intermediate had used Basic quite extensively during the early stages of their course. A final group of expert programmers consisted of subjects drawn from a population of teachers of programming and professional programmers employed in industry. None of the subjects in this group had less than 3 years post-degree programming experience, while a number of members possessed over 10 years post-degree experience.

Materials/Experimental Programs

Subjects were asked to produce programs from natural language specifications of three problems. One of these problems was the 'rainfall problem' (see figures 8.1 and 8.2) used by Johnson and Soloway (1985). The second problem involved determining whether an integer supplied as data was a prime number or not. The third problem specification was concerned with the calculation of a maximum and a minimum value from a series of numeric keyboard inputs.

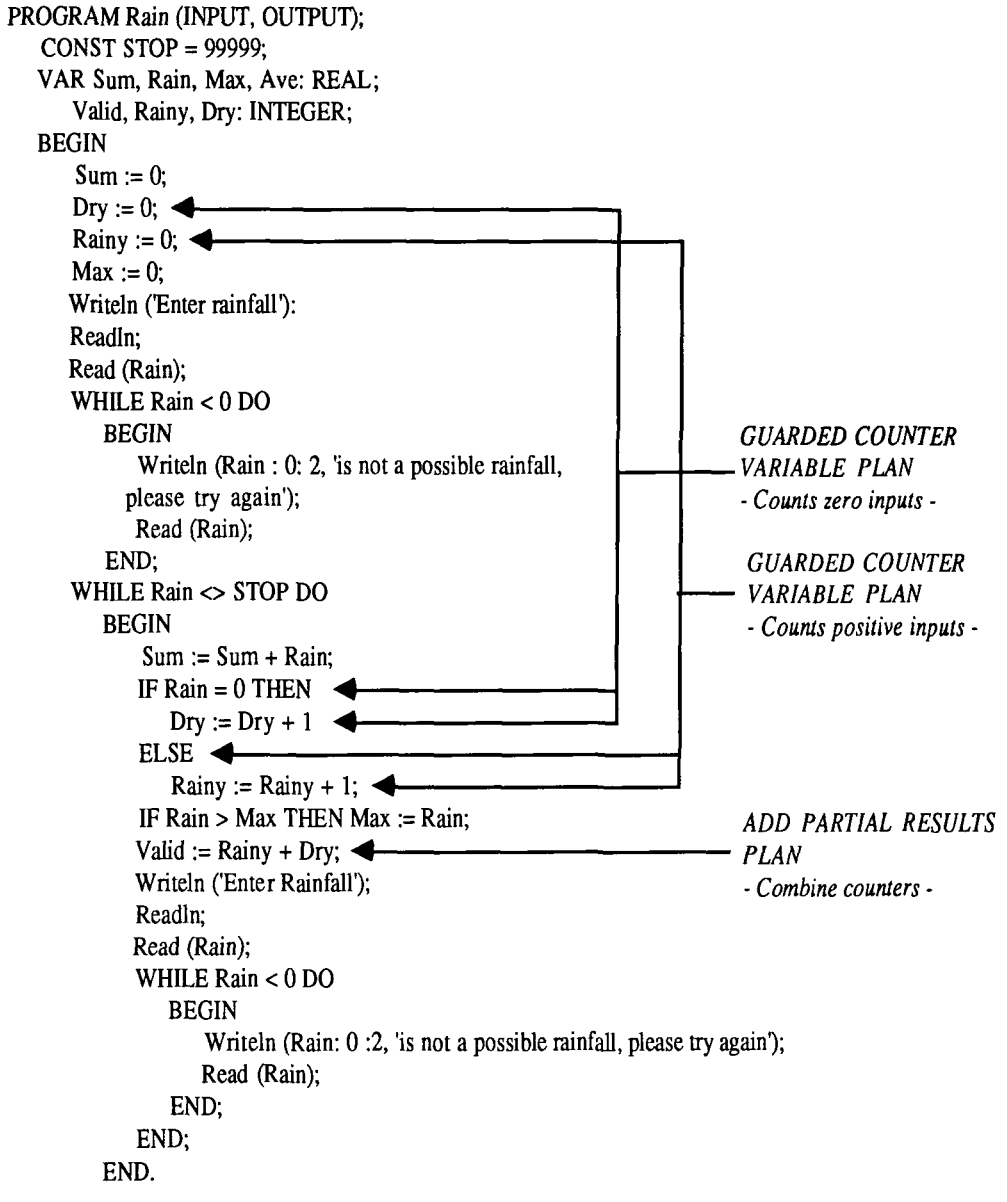


Figure 8.2. A standard Pascal solution to the rainfall problem with a number of plan structures illustrated (from Johnson and Soloway, 1985).

Design

The experiment employed a program generation task. There were two independent variables:-

1. Language - Pascal/Basic

2. Skill level - Novice/Intermediate/Expert

and two dependent variables:-

1. Number of jumps performed by jump-type classification (Inter/Intra-plan jumps)

2. Length of pause between jumps by jump-type classification (Inter/Intra-plan jumps)

8.4 Procedure

Subjects were presented with a short description of one of the three experimental problems and were asked to generate a solution using a familiar full-screen editor. They were allowed 5 minutes to complete this task. Subjects were not allowed to use pencil and paper but could make on screen notes if they wished. Subjects were asked to produce solutions in both Pascal and Basic, but the order in which they were requested to code their solutions in either language was randomised. All subjects attempted to generate solutions to all three of the experimental problems. The order of presentation of these problems was randomised. Transcripts of all on-screen activity were obtained for future analysis using the UMIST MMI monitor (Morris, Theaker, Phillips and Love, 1988). This device enables non-invasive recording of all user keystrokes and machine responses and provides controllable real-time (and half-real time) playback of user activities via the host machine. These transcripts were subsequently analysed for the presence of plan structures in code, the occurrence and nature of plan jumps and the pause duration between jumps.

A number of protocols were established in order to ensure a level of consistency within these different measures. The presence of plan structures was analysed by a group of three experienced programmers, all of whom were briefed about the nature of programming plans and were informed which plans might be expected to occur in each of the programs. These plans were derived from Johnson and Soloway (1985). Each member of the group analysed all the resulting program generation transcripts in terms of the expected plan structure of the program. They were asked to associate each line of the program with a specific plan. The raters were requested to carry out their analysis in terms of the plans identified by the experimenter. However, they were encouraged to suggest other plans within the program that were not made explicit in the initial plan analysis. It should be noted that no new plans were identified during this process. Figure 8.3 shows two program fragments illustrating comparable plan structures in Basic and Pascal.

Plan jumps were defined as follows: Intra-plan jumps were classified as movements between a current cursor position to positions within the same plan structure. Inter-plan jumps were classified as movements between a current cursor position to positions within different plan structures. These protocols applied to situations where new text was being inserted or existing text modified. Pause time between jumps was recorded in milliseconds, but this level of recording sensitivity was not thought necessary for the analysis. Hence, pause time is represented to the nearest second.

Edits to line numbers in Basic and to indentation structure in Pascal were excluded from the analysis since neither editing operation has a counterpart in the other language. It was thought that inclusion of these data in the analyses could give rise to difficulties in the comparison of plan editing between the two languages.

```
.
.
BEGIN
  Sum:=0;
  Rain:=0;
.
.
.
Read (Rain);
WHILE Rain <> 99999 DO
   BEGIN
      IF Rain<0 THEN
.
.
    ELSE
      BEGIN
         IF Rain=0 THEN
            Vaild := Vaild + 1
         ELSE
            BEGIN
               Vaild := Vaild + 1
               Rainfall := Rainfall + 1
            END;
         Sum := Sum + Rain
         IF Rain>Max THEN
               Max := Rain
      END

.
.
        Read (Rain);
   END;
Average := Sum/Valid;
```

Figure 8.3a A Pascal program fragment indicating a running total loop plan and an average plan

```
.
50 REM avrprob
.
.
80 LET Count = 0
90 LET Sum =0
100 REPEAT

.
.
140 INPUT New
145 IF New = 99999 THEN GOTO 170
150 LET Sum = Sum+New
160 LET Count = Count + 1
170 UNTIL New = 99999
.
190 IF Count = 0 THEN PRINT "No legal inputs" ELSE PRINT
                                "Average is..."; Sum/Count
.
```

Figure 8.3b). A Basic program fragment indicating a running total loop plan and an average plan

Figure 8.3. Two program fragments representing similar plan structures in Pascal and Basic. Note that these programs do not compute exactly the same function. This reflects the variation typically found in the subject's answers.

8.5 Results

Plan jumps

Figure 8.4 shows the mean number of inter and intra-plan jumps performed by subjects during generation by novice, intermediate and expert programmers using either Pascal or Basic. These data were entered into a three-way analyses of variance with the following factors in each case:

1). Skill level (novice/intermediate/expert)

2). Jump type (inter/intra-plan jump)

3). Language (Pascal/Basic)

No main effect of skill level or language was apparent. There was a significant interaction between jump type and skill level ($F_{2,132} = 6.34$, $p<0.01$). This appears to reflect the orthogonal relationship between inter and intra plan jumps with increasing levels of expertise. In addition, a complex three-way interaction between language, jump type and skill level was evident.($F_{2,66} = 3.72$, $p<0.05$). Separate ANOVAs for the results from each jump-type classification were employed in attempt to clarify the nature of this more complex interaction. These ANOVAs revealed that the skill level x language interaction was significant in the case of the inter-plan classification ($F_{2,66} = 8.43$, $p<0.01$) but not for the intra-plan classification. This interaction appears to be a consequence of the greater number of inter-plan jumps performed by intermediate and expert Pascal programmers in comparison to their Basic counterparts. There were no other significant main or interactional effects.

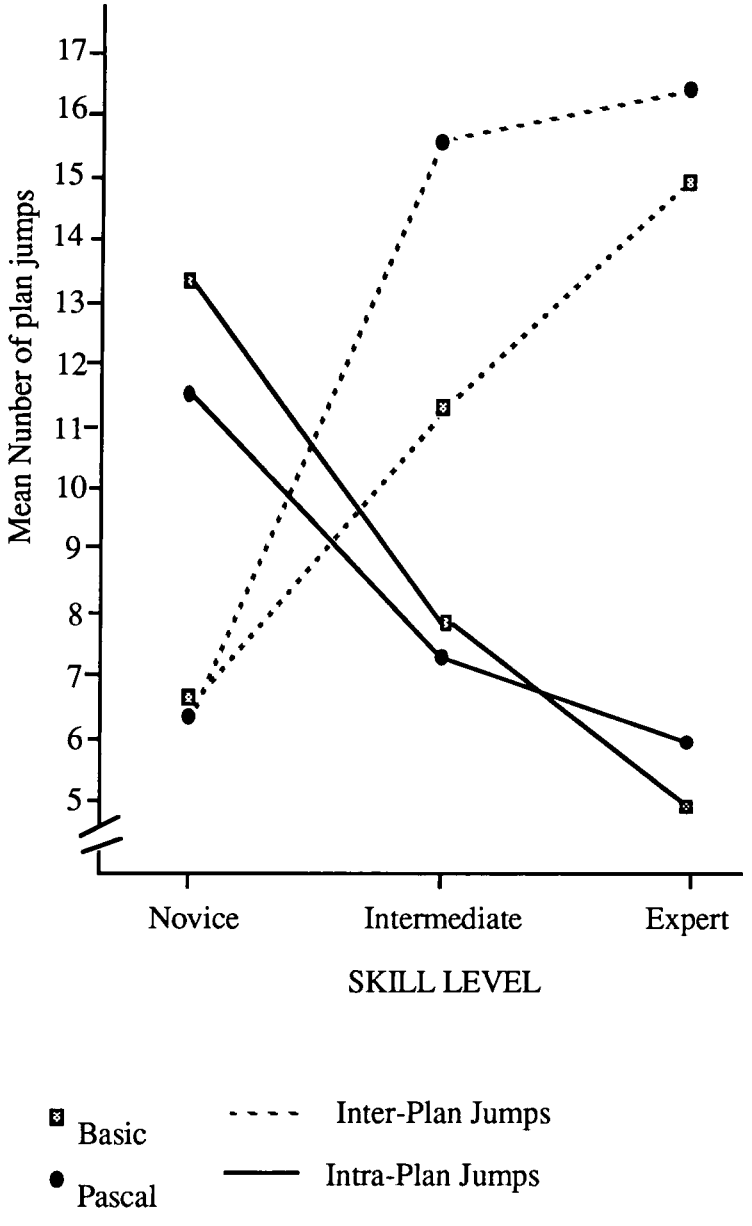


Figure 8.4 Mean number of inter- and intra-plan jumps performed by programmers of different skill levels in Pascal and Basic during program generation.

Pauses

Pause data (figure 8.5) was analysed in a similar fashion using a three-way ANOVA. The three factor levels were the same as above. This analysis indicated the following effects:

There was no main effect of either language or skill level. The effect of jump type was highly significant ($F_{2,132} = 18.2$, $p<0.01$). There was a significant interaction between skill level and jump type ($F_{2,66} = 5.89$, $p<0.01$) and a three-way interaction between language, skill level and jump type ($F_{2,66} = 10.74$, $p<0.01$). Again, separate ANOVAs for each jump type classification were carried out. This procedure indicated that the language x skill level interaction was significant in the case of the Inter-plan classification ($F_{2,66} = 7.55$, $p<0.01$) but non significant for the intra-plan classification.

Additional language comparisons

A comparison of the average number of plans generated for different programming languages (Basic or Pascal) revealed no significant difference between languages (t-test). The average length (lines of code) of each plan did not differ significantly between languages (t-test). In addition, the average length (lines of code) of Pascal and Basic solutions did not differ significantly (t-test). However, novices generated significantly fewer plans than both intermediates and experts, while a comparison of intermediate and expert performance indicated no significant difference in plan generation (t-test).

A measure of inter-judge scoring reliability was obtained in order to ensure a level of concordance between the three judges assessing programs for the presence of plans. A high coefficient of concordance was found to exist between judges (Kendall's coefficient of concordance $W = 0.77$, $p<0.01$).

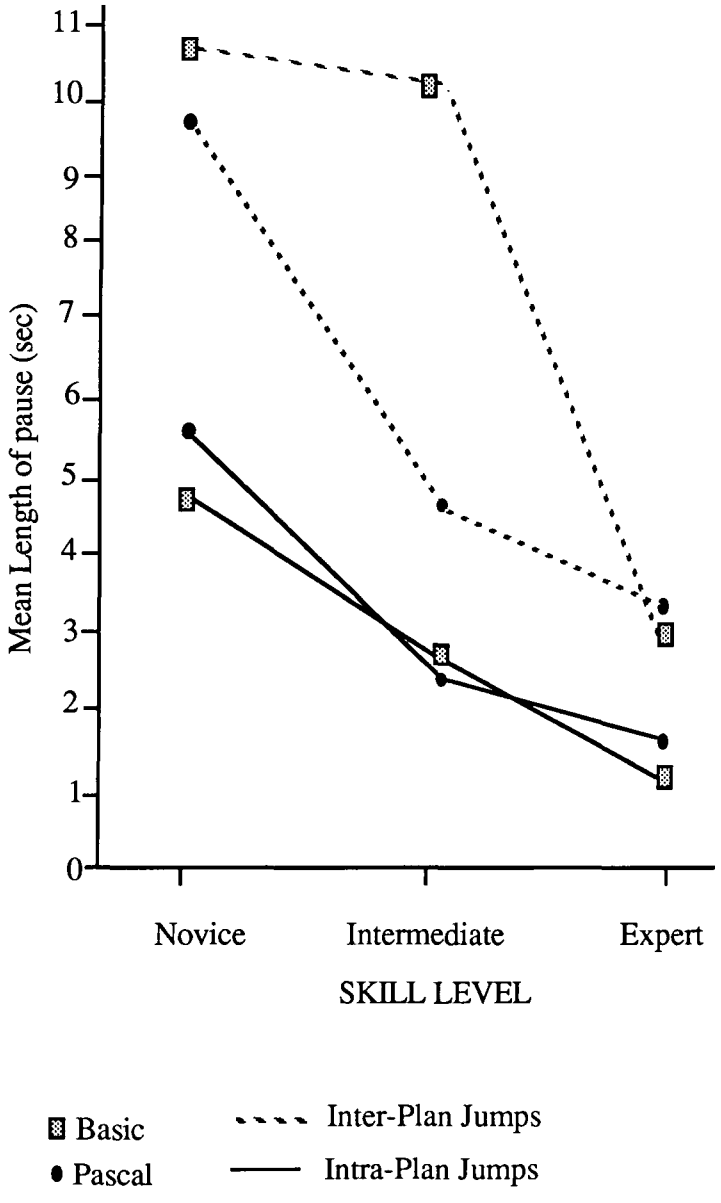


Figure 8.5. Mean length of pause (seconds) between inter- and intra-plan jumps for different languages during program generation

In addition, a plan analysis of all the resulting program transcripts was carried out by the experimenter in order to attempt to corroborate the results produced by the analysts. Both analyses were carried out independently and there was a high level of concordance between experimenter and judges analyses in terms of both the plans that were identified ($W = 0.57$, $p<0.01$) and their identification with particular lines of code in the program ($W = 0.64$, $p<0.01$).

The program transcripts were also analysed in terms of the comparative distribution of plans in Pascal and Basic. This analysis was undertaken because it is possible that the plans generated in one language might be implemented in a localised group of statements, and in another language be more widely distributed in the program, thus giving rise to problems interpreting the plan editing process.

Here, all lines in a program that corresponded to the same plan construct were identified. Each set of program statements (corresponding to a particular plan) were given the same label. Next the distance from the first line of a particular plan to other lines comprising that plan were assessed. Hence, if the next line of the plan was immediately adjacent to the initial line this was scored as zero, if it occurred on the next line it was scored as one, and so on until reaching the last statement of a particular plan. This procedure provides a broad measure of the distribution of the elements of a particular plan . An average indication of plan distribution in the two languages can be computed by summing these distribution measures for each plan and dividing this by the total number of lines comprising each plan. The average distribution measure for Pascal plans (0.11) did not differ significantly from the average distribution measure for Basic plans (0.09), [t-test]. The fact that this distribution measure is greater than zero for both languages does suggest some plan distribution, however this distribution is minimal and, by and large, plans appear to be generated in spatially contiguous blocks of code in both Pascal and Basic.

8.6 Discussion

Taken together these data suggest a rather complex relationship between skill level, language and programming strategy. While they provide support for particular elements within existing theories of programming, they also reveal some of the more complex interactions between these elements which are not predicted by such theories. For instance, as predicted by Rist (1986), the results of the present study indicate changes in the strategic use of plans as skill levels increase. Hence, the linear generation mode adopted by novices is replaced by a strategy that appears to reflect a development in plan focus as expertise increases. This is evidenced by the significant increase in inter-plan jumps with increasing expertise and a concomitant decrease in intra-plan jumps during the program generation task. These data also support a view of program generation that is similar to Jeffries' (1982) analysis of the strategies involved in reading programs. Jeffries found that experts read programs in the order in which they would be executed. In contrast, novices tend to read programs from beginning to end, in linear order like a piece of text.

In addition, the pause data suggests that as expertise increases the time spent pausing between both inter and intra-plan jumps decreases. This may reflect the type of speed-up function typically found in other studies of skilled performance within both a general problem solving context (Anderson, 1982) and also within the programming domain (Anderson, 1987; Wiedenbeck, 1985) . Also, evidence for plan boundaries is suggested by the interaction between jump type and skill level for pause data. Hence, the pause time between inter-plan jumps is greater than that between intra-plan jumps for all skill levels. As in other studies (Haberlandt, 1980; Reitman and Rueter, 1980; Robertson and Black, 1986) this suggests the existence of discrete plan boundaries in the programmer's knowledge representation.

The effects of language on generation strategy are however rather less straightforward. The model of coding presented by Green et al (1987) predicts a clear relationship between the notational features of particular languages and the development of programming strategies. Hence, if this model were correct, a language such as Basic should inhibit plan use because of certain features of its

notation while Pascal, which is thought to offer greater plan discriminability, should facilitate plan use. The results of the present study, however, suggest that language has little overall effect upon programming strategy. This concurs with results obtained by Bellamy and Gilmore (1990) which also failed to provide evidence for a straightforward relationship between language and programming strategy.

While the results of the present study do not reveal a main effect of language, they do indicate a potentially interesting three-way interaction between plan-type, language and skill-level. Further analysis suggests that this interaction results from the fact that a greater number of inter-plan jumps are performed by intermediate and expert Pascal programmers in comparison to their Basic counterparts. One reason for this seems to be that the effect of notation in the determination of programming strategy plays a greater role as programming skills develop, and particularly at intermediate skill levels.

Additional evidence for this interaction effect is to be found in the analysis of pause data during program generation. Once again no main effect language was evident, however the language x skill-level interaction was highly significant in the case of the inter-plan jump classification. From figure 8.5 it is clear that, for intermediates, the length of pause between inter-plan jumps in Pascal is significantly less than that occurring between inter-plan jumps in Basic. This may suggest that as programming skill develops the notation of the language (and its related plan discriminability etc.) has a significant effect upon plan use, but is of little relevance as a determinant of strategy at lower and higher skill levels.

The interpretation of this effect may be quite straightforward. For the novice, plan use is hypothesised to be minimal (as demonstrated in the present study), hence one would not expect notational factors to play a role. As expertise increases then the mean length of pause between inter-plan jumps for Pascal falls sharply. In contrast, there is only a slight reduction in the mean length of pause for the Basic data. Hence, for Pascal programmers, at the earlier stages of skill development, the notation of the language appears to support the use of plan structures.This is evidenced by the reduction in the mean length of pause which is taken to be an indicator of the ease with which plan structures can be used (selected or

implemented). The small decrease in the mean length of pause for Basic programmers at early stages of skill development is assumed to support the contention that the notation of this language does not support the use of plans.

At later stages of skill development the results indicate an inversion of the above results. Assuming a straightforward notational view we would expect to find a linear relationship between the ability to use plans and increasing levels of expertise, regardless of language. However, figure 8.5 illustrates that for Pascal programmers the rate at which the mean length of pause reduces between intermediate and expert skill levels is minimal. For Basic programmers, however, there is a decrease in the mean length of pause between these particular skill levels.

Once again, these results give credence to the idea that at early levels of skill development the notational aspects of Pascal support the use of plan structures, but once expertise has developed to a certain point the effects of notation become less important. For Basic, the development of the ability to use plan structures appears to be hindered during the beginning stages of skill development. However, the initial adverse effects of notation on plan use soon diminish as programmers reach sufficiently high levels of skill. Hence, the question that needs to be addressed is why plan use appears to be differentially affected by the notational aspects of the programming language at different skill levels.

Green et al (1987; and see chapter 4) suggest that because of the limitations of working memory, programmers need to make use of an external medium (eg the VDU screen) as a temporary store for code fragments as they are generated. The aim of Parsing is to recreate the original plan structure from these code fragments. In addition, features of the notation of the language may aid or hinder plan use.

The formal structure of the programming language is not the only determinant of strategy. According to Green et al, strategy is also affected by the user's knowledge of how to perform external tasks - by their knowledge representation. However, as we have seen in previous discussion of this work, no consideration is given to the way in which the notational features of the programming language and the programmer's knowledge representation might interact to determine strategy.

Green et al claim that their parsing/gnisrap model is a model of expert coding behaviour, but the results of the present study suggest that expert programming strategies do not appear to be affected by notational features in the way that the parsing/gnisrap model would predict. It appears from the data that the parsing/gnisrap model provides a reasonable interpretation of intermediate performance, where notation appears to influence programming strategy in the manner predicted.

The model would, however, also predict a similar strategy to prevail during expert performance. If we assume, not unreasonably, that intermediates and experts have similarly constrained working memory, then the only other factor in the parsing/gnisrap model, excluding notation, that might influence strategy would be the programmer's knowledge representation. Hence, it appears that for experts, features of their knowledge representation take precedence over notation as a determinant of strategy.

Different notations are likely to facilitate parsing to a greater or a lesser extent (Green, 1989). Hence, in terms of the above interpretation, experts appear to be able to make use of some feature of their knowledge representation for programs that enables them to parse for particular structures more readily. One way in which parsing might be facilitated is via the recognition of 'beacons' which serve to indicate the presence of particular program components. Wiedenbeck (1986a, 1986b) has demonstrated that experts can recall these key lines or beacons much better than novices. In addition, the presence of beacons in programs has been shown to facilitate program comprehension (Wiedenbeck and Scholtz, 1989). These studies suggest that expert programmers possess and are able to access a representation of programming knowledge which in some way reflects the saliency of these key lines.

The results of the present study tend to support these findings and in addition provide a means of examining the way in which programmers' knowledge representations may develop. These data also suggest some quite subtile interactions between developments in knowledge representation and general notational features of particular languages. Hence, as programming skill develops,

features of the programmer's knowledge representation appear to reflect the increasing role of 'beacons' and 'focal lines', etc., in the program comprehension and generation process.

Hence, intermediate programmers, whom we might suggest are still involved in the process of developing these particular features of their programming knowledge, are likely to be affected, to a greater extent than experts, by the ease with which they can parse existing program structures back into internal semantic representations. The results reported above appear to support this argument and, while not providing a basis for the rejection of existing models, clearly indicate the need for a framework which can allow for the integration of a range of different models in order to explain the richness of this particular form of behaviour.

8.7 General Conclusions

8.7.1 Changes in programming strategy with expertise

The experiment reported in this chapter has demonstrated that a range of factors may contribute to the determination of programming strategy. These factors include the development of particular features of the programmer's knowledge representation and the way in which the notation of a language might facilitate the parsing of code structures into cognitive representations and vice versa. The results of the present study provide evidence for changes in strategy which are associated with increasing expertise. In terms of existing work, there appear to be a number of ways of accounting for these differences in strategy.

Firstly, one of the outstanding features of expertise in programming, and in other domains appears to be the particularly opportunistic nature of preferred cognitive strategy for programming and other tasks. Hence, the programming activity cannot be characterised as purely sequential, rather it might be better construed as consisting of bouts of activity each of which involve the creation of code fragments. These fragments are, in turn, continually re-evaluated and modified in respect to the particular goal or subgoal currently under consideration. In addition, the development of code may be postponed at any time in order that the

programmer might direct their attention to other goals or subgoals, possibly in response to the recognition of previously unforeseen interactions between code structures. Opportunistic strategies of this nature have been highlighted in a number of previous studies concerned with both program design (Guindon, 1989; Guindon, Krasner and Curtis,1987; Ratcliff and Siddiqi, 1985; Siddiqi, 1985; Visser, 1988) and the coding activity (Green et al, 1987). The non-linearities observed in the present study, which appear to characterise both expert and intermediate program generation behaviour, may provide additional support for this opportunistic view of the coding process.

An alternative view of the strategic changes that appear to accompany increasing expertise might be to suggest that programming strategy is transformed from a depth-first novice strategy to a breadth-first expert strategy. This position is advanced by Jeffries et al (1981) who claim that novices adopt a depth-first approach to problem solving in programming; that is, they tend to expand only one part of a solution at progressive levels of detail. In contrast, experts tend to synchronously develop many sub-goals at the same level of abstraction before moving on to a lower level.

While the results of the present study provide evidence that programming strategy changes with increasing expertise, they do not provide a means of distinguishing a depth-first vs breadth-first view of the programming activity, such as that suggested by Jeffries et al (1981), from an opportunistic characterisation of expert programming strategy. However, this study does indicate various factors which appear to contribute to the adoption of such strategies or to the ease with which they might be supported.The framework advanced below suggests two central determinants of programming strategy and stresses the fundamental importance of their interaction in the development of strategy.

8.7.2 The role of knowledge representation in the determination of strategy

Firstly, it seems clear that the programmer's knowledge representation must support the representation of salient code structures. Such structures, which might be characterised as 'beacons' or 'focal lines', act as partial descriptions of

particular code fragments and provide reminders that a segment of a program may need completing at a subsequent stage. In addition, the development of these code structures appears to coincide with increasing expertise. This may suggest that as expertise develops, knowledge structures change such that the organisation of these structures reflects the increasing importance of 'focal lines' and 'beacons' etc. Similarly, in the domain of software design, Jeffries, Turner, Polson and Atwood (1981) found that although novices used the same general problem-solving methods as experts, they lacked skills in two areas: applying processes for solving subproblems, and effective ways of representing knowledge. Jeffries et al attribute this latter deficit to the inadequacy of the organising functions provided by the novice's immature design schema.

While the present chapter does not seek to deny the existence of generic declaritive representations of programming knowledge (i.e., programming plans), it does suggest the need to consider the development of an asymmetry in programmers' knowledge structures with increasing expertise. Hence, such knowledge structures appear to facilitate the representation of the 'focal' aspects of plans, while the necessary, yet minor and subordinate parts of plans are represented with less saliency. Here, we might the adopt the common view of natural language text comprehension (Bower, Black and Turner, 1979; Kintsch, 1974; Rumelhart, 1975) and conceive of programming knowledge as being represented in terms of hierarchically structured schemata, with focal plan elements achiveing prominance in each plan or schemata hierarchy.

This model of knowledge representation in programming will be elaborated later in this thesis (chapter 12). While this model clearly does not rule out a plan-based approach, it does suggest certain limitations for the plan theory of programming. For example, one implication of the present study is that it would appear that the plan theory does not provide an adequate basis for a theory of programming expertise. Hence, the expert programmer does not simply have more plans than the intermediate. Rather, the development of expertise might be better characterised as a 'fine-tuning' activity whereby the focal elements of plans are identified and the kinds of knowledge asymmetry we have discussed are established.

It is the case that novice programmers seem to be able to access fewer plans than both intermediates and experts. This may mean that the plans novices are able to access are poor matches to the current goal under consideration. Hence, novices may feel the need to correct these badly-matching plans immediately. This may give rise to the finding that novices tend to work on one plan at a time and to the associated prevalence of intra-plan jumps that has been observed in the context of novice behaviour. Conversely, experts, who have a greater range of plans may be able to find better matches to a current goal and hence may be more prepared to suspend the development of a particular plan until later. This view of the development of programming expertise would suggest that programmers gradually acquire program-specific plan constructs and as a consequence one would presumably expect intermediates to possess a greater range of plans than novices and a correspondingly smaller range than experts. However, the results of the present study indicate that experts and intermediates generate the same number and range of plans while novices, in comparison, generate significantly fewer plans. This provides additional support for other findings reported later in this thesis. In particular, intermediate and expert programmers appear to perform at the same level when asked to detect plan violations in programs, suggesting a similar level of plan knowledge. Novices, by contrast, are very poor at detecting plan violations (See chapter 7).

In the present context, we might suggest that while both intermediates and experts have access to the same number and range of plans, in the latter case the representation of these plans has become attenuated in order to reflect a growing recognition of the importance the focal aspects of these plans. In the final chapter of this thesis a framework for understanding programming behaviour is advanced which attempts to provide a richer account of the devlopment of knowledge representation in programming. Here it is suggested that the behaviour of expert programmers is governed largely by the implementation and comprehension of focal plan structures.

The results of the present study contribute to our understanding of the processes which underpin schema or plan development, and provides a framework for elaborating the relationship between the development of knowledge structures and expertise. This is likely to have implications for schema theory as it is applied in

other domains, since few studies have been concerned the detailed relationship between schema development and expertise. Where this relationship has been studied, expertise differences are normally explaned in terms of either the relative completeness or incompleteness of schemas (Lesgold et al, 1988) or in terms of the presence or absence particular schemas (Soloway, Adelson and Ehrlich, 1988). In the present context, emphasis is placed upon the way in which knowledge is structured rather than upon the presence of schemas or their relative completeness. This issue is considered in greater detail in chapter 10 where a study is reported that indicates the central importance of schema restructuring in the development of expertise.

8.7.3 Notation as a determinant of strategy

The second part of our framework is concerned with an analysis of the way in which programming language notation might support programming strategy. One of the more interesting findings of the present study is that notation does not appear to support an opportunistic or a breadth-first strategy to the same degree for programmers of different skill levels.That is, the effects of notation on strategy are less extensive for experts than for intermediates. In terms of existing theory (Green et al, 1987) this effect would not be predicted, since there is no reason to believe that features of the task language notation should provide differential support for programming strategy, regardless of a programmer's level of expertise.

One way to explain this differential effect might be to suggest that notation and knowledge representation interact very strongly to determine strategy. Hence, as representations of programming knowledge are in the process of development, as we suggest in the case of intermediates, then any additional means of facilitating programming strategy, such as might be provided by certain features of the notation, are likely to be of particular importance. At higher levels of skill, factors relating to the organisation of knowledge appear to play a greater part in the determination and the support of programming strategy.

In addition, the work reported here suggests a slightly different model of planning to those models which are generally advanced as descriptions of the human planning activity. Most extant models of planning embody some abstract notion of the planning activity in which the nature of the problem representation is not shown to have an effect upon the kind of planning or problem solving strategy that is invoked. However, within the programming domain it is clear that features of the notation of the task language can facilitate, or indeed act to constrain, the preferred cognitive strategy that is adopted for this task. This is likely to have nore general implications for the study of planning within the range of domains that use formal or semi-formal notations to describe aspects of the problem space. Consequently, it might be suggested that there are certain dangers in attempting to divorce planning models from an analysis of the way in which tasks might be represented.

8.8 Summary

The framework advanced here suggests a number of implications for our understanding of the determinants of programming strategy. In particular, it is clear that the role of both notation and knowledge representation cannot be fully explained in isolation. This is because these factors appear to interact very strongly to determine programming behaviour. Hence, it is only through an analysis of these more complex interactions that a comprehensive elaboration of the determinants of programming strategy will be forthcoming. This chapter suggests the need to consider these interactions within a developmental framework. That is, within a context which views the development of programming skill as accompanied by subtle changes in the way in which programming knowledge is represented. This 'fine tuning' of programming knowledge appears to provide support for particular forms of preferred programming strategy. The use of such strategies is in turn assisted by features of the language notation, but only during the beginning stages of their development. As programming skill increases the role of notation appears to take less precedence as a determinant of strategy.

This analysis suggests that previous studies which have examined the nature of programming strategy only provide an interpretive basis within a rather limited

context. Hence, such studies, while emphasising the need to consider both notation and knowledge representation, have failed to elaborate the relationships and interactions between these factors within the general realm of skill development in programming. This chapter attempts to build upon the results of these studies, whilst at the same time stressing the fundamental nature of the interactions between these central determinants of strategy. A comprehensive understanding of the strategies involved in a complex task such as programming is only likely to be facilitated if we can not only delineate the role of its individual components, but also understand in detail the intricate nature of their interactions.

Chapter 9. Delineating forms of strategy and their relationship to the development of expertise in programming.

9.1. Introduction

In the previous chapter an attempt was made to demonstrate the way in which strategy may change with increasing levels of programming expertise. However, the experiments reported there did not allow for a clear differentiation between general forms of program generation strategy. Specifically, the findings reported in the last chapter did not enable a distinction to be made between breadth-first strategies and depth-first/opportunistic strategy. For that reason, the present chapter aims to provide a more detailed analysis of the generation strategies exhibited by novice and expert programmers.

The primary intention of this chapter is concerned with explicating the detailed relationship between the knowledge restructuring process that is proposed in this thesis and the emergence of particular forms of strategy. In particular, it has been suggested that this restructuring process improves the accessibility of particular elements of individual program schemata or plans. Moreover, it was suggested in the previous chapter that elements which are prominent in the schemata hierarchy are likely to be those which are salient to the programmer.

Extending this analysis further, Rist's concept of 'focal line' was adopted in order to provide some means of equating this psychological saliency with specific program structures. In terms of this analysis, it was hypothesised that focal lines will tend to be generated first during coding, and that programmers, having mapped out the general structure of the program at a high level of abstraction, will return later to these focal structures in order to elaborate the code surrounding them. This prediction is based upon the parsing-gnisrap model, however the present analysis endeavours to extend this model by both describing the salient aspects of a programmer's knowledge representation and by demonstrating the way in which this form of representation can affect programming strategy

In order to carry out such an analysis, an attempt has been made to define a number of explicit levels of abstraction in program structure. This is based partially upon Rist's model of focal expansion that was described in chapter 3. Additionally, emphasis has been placed upon the derivation of general behavioral regularities from data generated from a reasonably large number of subjects. Data has been collected from a retrospective analysis of code generation rather than from verbal protocols, as in Rist's studies, and this means that the imprecision normally involved in classifying salient behavioral aspects of the programming activity can be avoided. Collecting data in this way should reduce the inaccuracy stemming from the linearisation effects that are common in subjects' verbalisations about knowledge structures that have a significant temporal and/or spatial dimension (Levelt, 1981). Such effects are likely to be of particular significance in studies which use verbal protocols to provide evidence of particular forms of strategy. In the context of the present study, however, abstraction levels can be related explicitly to fragments of program code as they are generated. Hence, the study relies neither upon concurrent verbalisation nor upon potentially imprecise protocol classification schemes.

9.2 The programming task: Nonlinearities and focal expansion

Rist (1989) has proposed a model of program generation which traces the evolution of a program through a number of stages (see chapter 3). An explicit feature of Rist's model concerns the identification of levels of abstraction in program structure. It is claimed that programs are built from simple knowledge structures that are merged and combined to form more complex structures. Rist is primarily interested in the processes that underlie the plan generation activity and central to his theoretical explanation is the idea of focal expansion.

Focal expansion describes the process of generating a programming plan from a so-called 'focal line'. In terms of Rist's account, each programming plan has an associated focal line that directly encodes the goal of that plan. For instance a 'running total loop plan' will be associated with the focal line 'count:=count+1'. The complete plan will also consist of an initialisation component and some means of reading data values into the plan. The production of a program is seen to progress through various stages beginning with the implementation of a focal line,

its extension to form a complete plan and finally to the creation of an entire program through a process of plan merger.

The emphasis of Rist's model is concerned with the process of plan creation through focal expansion, however little attention is paid to the process of plan merger.This raises some important issues in the context of the present study. For instance, are plans created in a linear order such that the programmer completes one plan before moving onto the next? Or, conversely, are the focal lines of plans instantiated first to provide an abstract skeletal program structure which can later be extended to include less salient plan elements? The model of knowlege restructuring presented in this thesis would predict the latter, since focal lines are taken to represent a single level of abstraction within a program structure. Non-focal lines simply extend this plan focus. According to Rist (1989) "The (plan) focus ... marks the start of detailed design in the domain of the program" (p 403). The emphasis of Rist's model is, however, too constrained (considering, in detail, only the move from focal line to programming plan) to explain possible interactions between higher-level structures.

The present study attempts to address the issues outlined above by examining the evolution of high and low-level program structures during a program generation task. Information obtained from this analysis will provide some indication of the form of strategy that is adopted for this task.

Following Rist, it is assumed that focal and non-focal lines represent discrete levels of abstraction within the program hierarchy. One implication of the model presented in this thesis is that focal lines will be created before non-focal lines. Hence, one level of the program hierarchy will be established before lower-levels in the hierarchy are expanded.

Another way of investigating these issues is to explore more explicitly the nature of the nonlinearities found to exist in code generation. In the previous chapter an experiment was reported which explored these nonlinearities in the context of plan generation in programming, but an analysis of nonlinearities can also provide more specific evidence for the development of particular forms of strategy. In the present context, interest is directed towards the nonlinearites occurring both within and between hierarchical levels in program structure. Hence, a number of different

categories of nonlinearity are possible; jumps between a focal line and another focal line; jumps between a non-focal line and other non-focal lines (within the program's hierarchical structure). Conversely, jumps may occur between hierarchical structures; from focal line to non-focal line or vice versa.

Although the present study is primarily concerned with the generation strategies employed by experts, it has also been possible to investigate novice behaviour within this context. Hayes-Roth and Hayes-Roth (1979) suggest that expertise may influence the kind of planning model a planner or problem solver brings to bear on a particular problem, however no further consideration is given to this conjecture. Jeffries et al (1981) have shown that novices tend to adopt a depth-first approach to design problem solving while experts favour a breadth-first approach. Larkin et al (1980) have demonstrated that experts and novices employ different strategies when solving physics problems. Experts tend to work forwards from general physical principles to problem goals while novices work backwards from the problem goal. With these notable exceptions, many previous studies looking at both program tasks and at more general problem solving activities have failed to elaborate the mechanisms that underlie the relationship between expertise and the use of particular forms of strategy.

The model presented in this thesis makes two specific claims about the strategies that will be adopted for program generation tasks. In the case of experts, a top-down or breadth first strategy should be evident since it is suggested that programs will be generated in a manner which reflects the hierarchical structure of the programmer's knowledge representation. That is, one level of program abstraction should be mapped out before other levels are elaborated. Specifically, we have suggested that those structures representing focal lines will tend to be generated first and that these will later be elaborated to include less salient elements at other levels of abstraction. In contrast, it has been suggested that novice and intermediate programmers do not typically posses this differentiated form of programming knowledge. Hence, for these groups it might be predicted that generation will be in schema order, and that this will be reflected in the adoption of depth-first or opportunistically orientated strategies.

9.3 Participants, Procedure and Tasks

Forty subjects participated in this study. Subjects were split into two groups of equal size. One group of subjects were classified as experts and the other as novices. The expert group consisted of programmers/designers with a number of years industrial experience (4 to 13 years. Mean, 5.6 years) and of teachers of programming and software design, all of whom possessed previous industrial software design experience. The novice group comprised a number of Second year undergraduate students. Members of this group were drawn from the same student cohort and all had been instructed in the basic principles of traditional software design practice and structured programming.Subjects in both groups had experience of the programming language employed for this study - Pascal. All members of the expert group either taught Pascal or used it extensively in their work, while all members of the novice group had attended a first year course in Pascal and had used the language for project work.

Participants were asked to undertake a number of programming tasks of varying difficulty. The simplest problem (derived from Johnson and Soloway, 1985) required participants to construct a program that would calculate an average and a running total from a series of input values. More difficult problems were based upon Ratcliffe and Siddiqi's (1985) traffic counting task and a task derived from Rist (1989) which required participants to sort 2113 weights into ascending order. These problems might be considered to be fairly straightforward by professional standards, however it is clear that they demand a significant degree of problem solving behaviour - the more difficult tasks taking between 43 and 78 minutes for experienced programmers/designers.

Participants were provided with short natural language specification of the problems and were asked to write a program to solve each problem. Participants were told that they could make on-screen notes if desired, but were requested not to use pen and paper. No time limit was imposed on the tasks. Participants were asked to type their programs onto a familiar full-screen editor. All on-screen activities were recorded using the UMIST HIMS tool (Theaker et al, 1989); a non-invasive recording and replay device providing a number of analysis facilities.

```
[ ].
[ ].
[ ].
[I]  Sum:=0;                                  {Running Total Variable Plan}
[I]  Rain:=0;                                 {Guard Plan}
[ ].
[ ].
[ ].
[R] Read (Rain);                              {Guard Plan/Running Total Loop
Plan}
[E]  WHILE Rain <> 99999 DO                   {Running Total Loop Plan}
[M]    BEGIN
[E]       IF Rain<0 THEN                      {Guard Plan}
[ ].
[ ].
[M]    ELSE
[M]       BEGIN
[E]          IF Rain=0 THEN                   {Guard Plan}
[F]             Vaild := Vaild + 1            {Counter Variable Plan}
[M]          ELSE
[M]             BEGIN
[F]                Vaild := Vaild + 1         {Counter Variable plan}
[F]                Rainfall := Rainfall + 1   {Guard Plan}
[M]             END;
[F]          Sum := Sum + Rain                {Running Total Variable/Loop Plan}
[E]          IF Rain>Max THEN
[E]                Max := Rain
[M]       END
[E]      Writeln ( 'Please enter next value:' );
[ ].
[ ].
[R]       Read (Rain);                        {Running Total Loop Plan}
[M]    END;
[F] Average := Sum/Valid;                     {Average Plan}
[ ].
```

Figure 9.1. Program except illustrating plan structures and statement categorisation. [F] = Focal Line; [R] = Read statement; [E] = Extension; [I] = Initialisation and [M] = Miscellaneous.

The resulting programs were analysed for the presence of common plans by three independent raters using a goal hierarchy (Bellamy and Gilmore, 1990; Gray and Anderson, 1987). The plans that were identified varied little between the raters. For each plan, the raters were asked to identify the focal line of that plan based upon the definition provided by Rist (1989). Once again there was high degree of agreement between raters regarding the identification of focal lines. Figure 9.1 presents a program except with plan structures highlighted and illustrates an example of the coding scheme used to classify program statements. This coding scheme allows each statement to be categorised either as a focal line (a line that directly implement a current goal) or as a non-focal line. Non-focal lines may consist of statements representing a focal line extension, a plan initialisation or a read process (see Rist, 1989). All other statements were classified as miscellaneous.

From the above analysis, and by replying on-screen activity, a retrospective analysis of the temporal distribution of focal and non-focal lines was undertaken. In addition, a number of nonlinearities could be classified. For this purpose a nonlinearity was defined as a jump between one line of code to another. This could be either to edit an existing line or to insert a new line. An analysis was undertaken of the number of nonlinearities occurring between focal lines; between non-focal lines and between focal and non-focal lines and vice versa.

9.4 Results

Figure 9.2 shows the mean number of focal and non-focal lines (representing different abstraction levels) generated by both novice and expert programmers during the experimental session. The number of lines generated within each block consists of a simple count of the lines produced (focal or non-focal - according to the protocol outlined above) during each 10 minute time period, and is therefore not a cumulative count over the entire session.

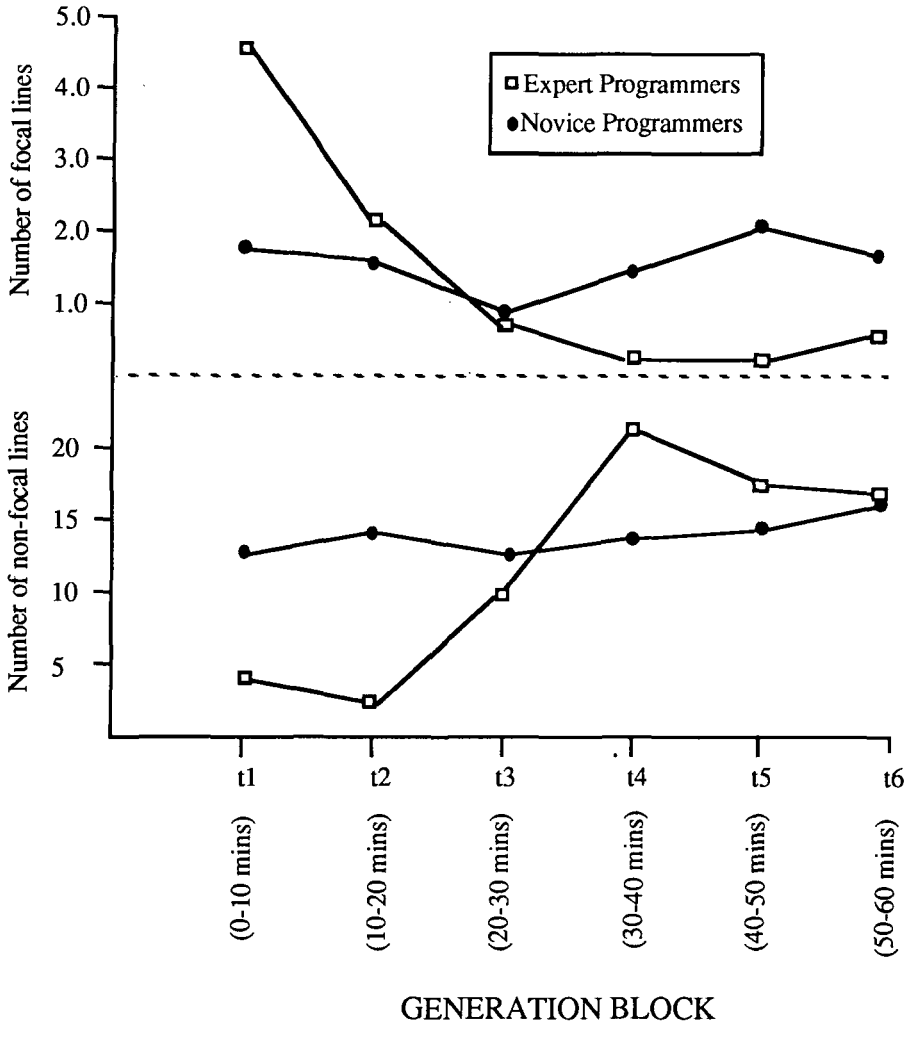


Figure 9.2. Mean nunber of focal and non-focal lines generated by expert and novice programmers during each generation block.

The cumulative mean number of focal lines generated by experts and novices over all generation blocks did not differ significantly (t-test) (mean[Novices] = 10.4: mean[Experts] = 9.4). Similarly, no difference was evident between the cumulative mean number of non-focal lines generated by these groups over the same period (t-test) (mean[Novices] = 79.1 : mean[Experts] = 72.3).

These data were entered into a pair of two-way ANOVA's - one constructed with the focal line data, and the other with non-focal line data. In each case the two factors were:

- Group (Expert/Novice)

- Generation Block (t_1 through t_6)

For the focal line data no significant main effects were evident. However, a significant interaction between generation block and group ($F_{5,228} = 6.73$, $p < 0.001$) was apparent. This interaction appears to reflect the decrement in focal line generation that can be observed in the case of the expert group as the session progressed. The novice group, in contrast, appeared to maintain a fairly constant rate of generation throughout the course of the session. Further support for this finding was provided by instituting multiple pairwise comparisons of means between all adjacent generation blocks using the Newman-Keuls test with a significance level of $p < 0.01$. In the case of the expert group, significant differences were found to exist between blocks t_1 through t_4, with no significant differences between t_4 through t_6. For the novice group there were no significant differences in an identical range of *post hoc* comparisons.

Similar findings emerge from the non-focal data. Again, no main effects were apparent. However, there was a significant interaction between generation block and group ($F_{5,228} = 10.32$, $p < 0.001$). Multiple *post hoc* comparisons of means indicated significant differences between blocks t_2 through t_4 for the expert group, reflecting an increasing rate of non-focal line generation during this period. No significant differences were evident for all other *post hoc* comparisons.

Figure 9.3 shows the number of nonlinearities occurring between and within hierarchical level. For this purpose a between hierarchy jump was classified as a jump between a focal line and a non-focal line or vice versa and a within hierarchy jump as a jump between a focal line and another focal line or between a non-focal line and another non-focal line. These data were analysed using a identical procedure to the one reported above.

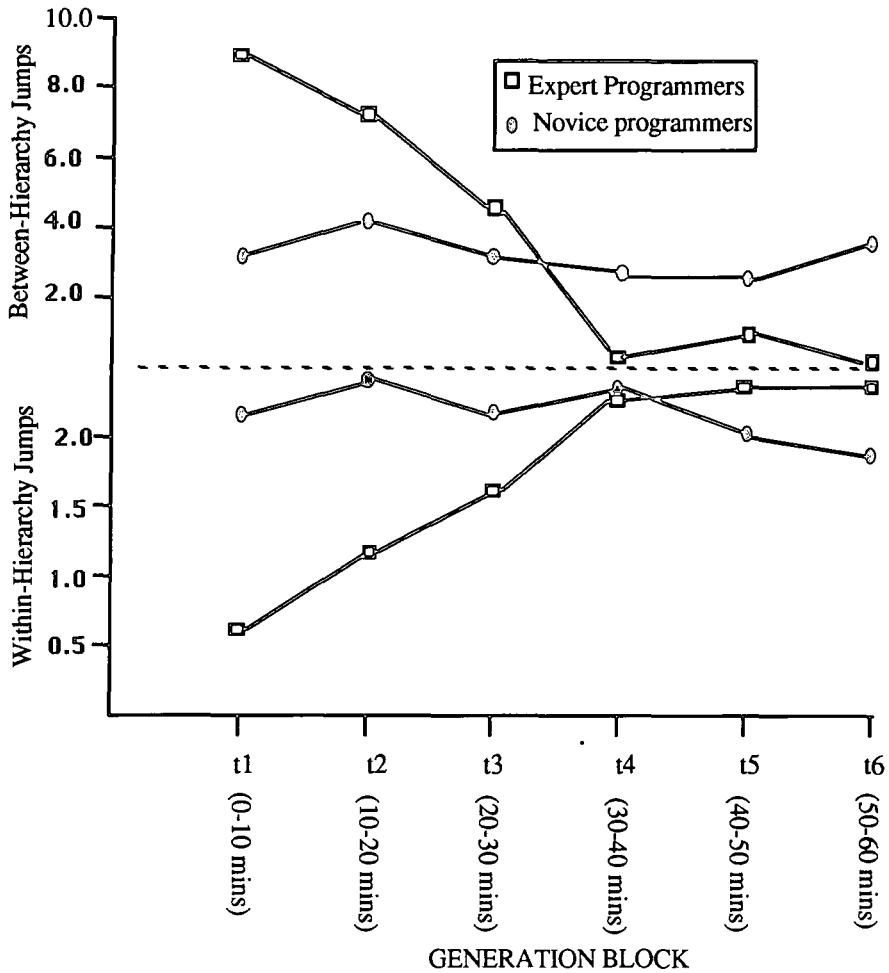


Figure 9.3. Mean number of Between and Within-Hierarchy jumps performed by novice and expert subjects during each geneartion block.

The cumulative mean number of within hierarchy jumps for experts and novices over all generation blocks did not differ significantly (t-test) (mean[Novices] = 12.5: mean[Experts] = 10.2). Similarly, no difference was evident between the cumulative mean number of between hierarchy jumps for these groups over the same period (t-test) (mean[Novices] = 20.9: mean[Experts] = 21.3).

These data were entered into a pair of two-way ANOVA's - one constructed with the within hierarchy data, and the other with between hierarchy data. Again, the two factors in both analyses were:

- Group (Expert/Novice)

- Generation Block (t_1 through t_6)

In the case of the between hierarchy data no main effects were apparent. There was, however, a significant interaction between Group and Generation block ($F_{5,228} = 4.23$, $p < 0.001$), reflecting the decreasing rate of between hierarchy jumps over generation blocks for the expert group. Multiple *post hoc* comparisons reveal significant differences ($p<0.01$) between means for blocks t_1 through t_4 for the expert group. No other *post hoc* comparisons proved to be significant.

For the within hierarchy data, there was an interaction between Group and Generation block ($F_{5,228} = 5.89$, $p < 0.001$). Here, multiple *post hoc* comparisons indicated significant differences between means for blocks t_1 through t_5 for the expert group. Again, no other sigificant differences were apparent in the case of the novice group in an identical range of comparisons.

9.5 Discussion

These results clearly have a number of implications for the way in which we might attempt to characterise the program generation activity. Firstly, the strategy adopted by experienced programmers appears to broadly correspond to the adoption of a top-down, hierarchically levelled approach. Hence, the abstract structure of the program, represented by the instantiation of focal lines, is mapped out at an early stage in the evolution of the program. This high level structure provides a framework around which the rest of the program can be built. However, at many points during the evolution of the program, programmers can be seen to engage in opportunistic behaviour - synchronously generating both focal and non-focal structures. Hence, at any particular point during the programming activity, behaviour might legitimately be described as opportunistic. However, this clearly does not rule out the existence of a more global top-down programming strategy.

At early stages during program generation, expert programmers make many jumps between structures at the same level of abstraction. As the task progresses, however, nonlinearities occur between hierarchical levels and strategy is seen to become more opportunistically oriented. This corresponds to the findings that emerge from the analysis of focal/non-focal line data reported in the first study.

Taken together, these findings clearly demonstrate that different forms of strategy may be adopted within the context of a single task, and further, that choice of strategy may not be primarily determined by task characteristics as suggested by existing work (Carroll et al, 1980; Hayes-Roth and Hayes-Roth, 1979). The results reported here indicate that during the early stages of the development of a program, opportunistic episodes are subsumed within a more general top-down approach. However, during later stages, program generation takes on a more opportunistic character as programmers begin to perform significantly more jumps between distinct hierarchical levels.

A rather different picture emerges in the context of novice behaviour. Less experienced programmers appear to display highly opportunistic behaviour throughout the course of the programming activity. This finding is rather surprising since all the programmers participating in this study had some training and experience (albeit rather limited in the case of the novice programmers) in top-down programming methodologies. One might reasonably expect this to be reflected in the generation strategy that is adopted. However, in spite of this, novice programmers appear to maintain a fairly constant generation rate for both focal and non-focal lines and tend to perform as many jumps between hierarchical level as within. Such behaviour could only be described as opportunistic as it reflects none of the characteristics that are implied by top-down models of programming.

The mechanisms which might be thought to underlie the differences that have been found to be associated with different levels of expertise are not clear. Anderson (1983) suggests that behaviours which might be described as opportunistic deviations from hierarchical problem solving may often arise as a consequence of fairly simple cognitive failures. For instance, subjects may pursue details of a current plan that is inconsistent with higher level goals simply because they have forgotten or have misremembered these goals. Both Hoc (1988) and Guindon et al

(1987) have observed that even expert designers may experience difficulties simultaneously considering all possible solution elements at a single level of abstraction, thus leading to an opportunistic approach as opposed to a the adoption of a strict top-down strategy.

From this we might conclude that opportunistic episodes arise largely as a consequence of simple cognitive failures. By and large these failures appear to be related to working memory capacity limitations. For instance, this is apparent from the difficulties experienced by programmers in simultaneously maintaining all possible solution elements at one level of abstraction. Opportunistic strategies may be a symptom of these kinds of failure or alternatively it may be adopted as a deliberate means of circumventing working memory limitations in order to reduce the frequency and scope of possible failures. In the context of the present study it is not possible to resolve this issue. However, the findings of a study reported in chapter 11 of this thesis looking at the role of working memory and expertise in programming shed some light upon the prevalence of opportunistic episodes in programming behaviour.

Here it has been observed that expert programmers tend to rely extensively upon the use of external memory sources (VDU screen, notes on paper etc) when generating a program. Conversely, novices rely to a much greater extent upon the use of internal memory to develop as much of a solution as possible before transferring it to an external source. The reasons for this are as yet unclear. However, such a strategy may give rise to the opportunistically oriented behaviour evident in the case of novice programmers in the context of the present study. Relying in this way upon internal memory sources means that novices will experience difficulty simultaneously maintaining aspects of an emerging program in memory. In particular, it will prove difficult to map out a global framework at a single level of abstraction - that is, adopt a hierarchically levelled approach.

Indeed, Rist (1989) has found that novice programmers tend to adopt a very localised coding strategy, which places a minimal load on their working memory, and design the same part of a solution at different levels of abstraction. In contrast, experts appear to employ a strategy that involves greater use of external media to reduce working memory load. The present study suggests that expert programmers develop solutions in a hierarchically levelled fashion and this may

only be possible if an external media is used to record partial fragments of the solution as it develops. However, as the program evolves, experts seem to revert to the more localised strategy that is evident in the case of novice behaviour, and tend to engage in a greater frequency of opportunistic episodes.

One possible reason for this phenomenon may be that, in the case of expert performance, there is a clear temporal separation between the progressive and evaluative problem solving activities that are normally involved in the programming task. Gray and Anderson (1987) suggest that problem solving activities in programming can be described as either progressive (working directly toward the goal state of a problem) or as evaluative (evaluating some already executed part of the problem solution). One might speculate that expert programmers will tend to engage broadly in progressive activities during the early stages of a programming task and in evaluative activities toward its latter stages.

Hence, in the context of the present study, we might characterise the early top-down stages of program generation as progressive and the later opportunistic activities as arising from an evaluative process which has led to the location of flaws in the emergent program. For novices, progressive and evaluative episodes may be more closely linked and this is likely to be manifested in temporally localised generation and monitoring activity. This may give rise to a more systematic pattern of opportunistic episodes as small localised parts of an emergent program are evaluated and then modified when necessary. In order to provide additional support for this hypothesis, an analysis of the temporal distribution of evaluative and progressive activities in a programming task is presently being undertaken. This analysis is not yet complete. However, early indications appear to provide support for notion that differences in expertise are associated with distinct temporal patterns of progressive and evaluative activities such as those discussed above.

9.6 Conclusions

The studies reported in this paper differ in a number of significant respects from existing work which has examined salient behavioral characteristics of the programming activity. In particular, the use of a comparatively large subject

group means that the possible confounding factors which might arise as a consequence of intra-subject variation are minimised. In addition, it has been possible to derive, from previous empirical work (Rist, 1989), a number of distinct levels of abstraction within the problem space upon which to base a behavioral analysis.

A number of interesting findings emerge from this analysis. Firstly, it has been shown that opportunistic episodes may occur at any point during the evolution of a program. However, this does not rule out the existence of an overall top-down strategy. Hence, the clear dichotomy between top-down and opportunistic approaches that is implicated in previous work may be unfounded. It is unlikely that studies involving an analysis of the behaviour of a relatively small number of subjects would make the observation of these regularities possible.

Secondly, it has been shown that the emergence of top-down or opportunistic strategies is not task dependent as suggested by a number of previous studies. Rather such strategies can co-exist within the context of a single task. However, one form of strategy make take precedence over the the other at particular points during the evolution of the program.

Expertise also appears to play a major role in the determination of particular forms of strategy. For instance, novice programming behaviour appears to be systematically opportunistic, displaying none of the characteristics of a top-down approach. Conversely, expert programmers adopt a broadly top-down approach, at least during the early stages of program generation. In this context, the notion of the focal line appears to play a significant role in expert programming behaviour. The experiment reported in this chapter has shown that experts tend to generate focal lines first, providing a framework around which the rest of the program can be constructed. This may arise as a consequence of the knowledge restructuring process proposed in this thesis whereby the devlopment of expertise is seen to be accompanied by the restructuring of plan/schemata structures such that focal lines achieve promanance.

Finally, there appear to be clear differences between the temporal distribution of opportunistic and top down episodes when comparing novice and expert programming behaviour. It has been proposed that the pattern of this distribution may be indicative of differences existing between experts and novices in terms of the separation of progressive and evaluative programming episodes. It appears that novices adopt a very localised evaluation strategy that is likely to give rise to localised changes, and consequently to a systematic temporal distribution of opportunistic behaviour. Conversely, experts tend to engage in evaluative episodes toward the latter stages of the programming task, giving rise to an asymmetrical pattern of top-down and opportunistic behaviour.The idea that differences in expertise are associated with distinct temporal patterns of progressive and evaluative activities is again only speculative. This hypothesis will require further empirical support before any firm view can be established.

In conclusion, this chapter suggests a rather different view of the programming activity to that proposed in previous studies. In particular, top down and opportunistic strategies are seen to co-exist within the context of a single task. Additionally, there appear to be clear differences between novice and expert programming strategies. The psychological mechanisms that may underlie these differences are as yet unclear, but further empirical studies will hopefully provide a foundation for a more detailed analysis and specification of these mechanisms. In particular, future experimental work should address the distribution of progressive and evaluative episodes in program generation and the relationship of this distribution to differences in expertise. This will provide more information on the behavioral aspects of the programming activity and may contribute to our understanding of more general problem solving tasks.

Another issue that will need to be addressed relates to the generalisability of the present study. Programming is clearly taught in a top-down fashion and many views of the program design process prescribe a top-down, hierarchically levelled approach (see chapter 5). It may be the case that a rational top-down process of this nature is not appropriate to the software design activity (Parnas and Clements, 1986). Such a view of the design process may stem from the tendency of previous work to conflate descriptive and prescriptive accounts of the design activity (Carroll and Rosson, 1985). However, the effects of the way in which programming is taught are likely to be closely related to the way in which program

ming actually takes place. Hence, it is difficult to disentangle 'natural' preferred programming strategies from those which are acquired and/or prescribed.

Chapter 10. Knowledge restructuring in programming: Evidence for salient psychological structures derived from reaction time and errors.

10.1 Introduction

This chapter explores in greater detail the relationship between knowledge structure and organisation and the development of expertise in programming. An empirical study is reported which provides support for the model of knowledge organisation in programming that is presented later in this thesis (see chapter 12). This model stresses the importance of knowledge restructuring processes in the development of expertise. In the context of the present chapter, this is contrasted with existing models which have tended to place unique emphasis upon plan or schemata acquisition as the fundamental mode of learning associated with skill development in programming and other domains (see chapter 3). For example, it is clear from our earlier discussion of the plan theory of programming presented by Soloway and others, that the development of expertise in programming depends largely upon the acquisition of plans. This work not only neglects to consider the detailed structure of plan knowledge, but also provides no indication of the way in which plan structures are implemented as programs.

10.2 Schema theory and expertise

Schema theory provides the theoretical foundation for a great deal of work in contemporary cognitive science (Schank and Abelson, 1977; Rumelhart, 1975).The idea that understanding and comprehension are mediated by schematic stereotypical knowledge structures extends a tradition begun by Bartlett (Bartlett, 1932) and the Gestalt psychologists earlier this century. Contemporary schema theory now underpins accounts of text comprehension (Kintsch and van Dijk, 1978) and complex problem solving (Larkin, 1983; 1985) and has been applied

more recently to a variety of HCI domains including text editing (Kay and Black, 1984) and computer programming (Soloway and Ehrlich, 1984).

The application of schema theory to our understanding of problem solving has proved important for a number of reasons. Firstly, it provides a means of describing fundamental aspects of a problem solver's knowledge representation for a complex task. From this it should be possible to derive predictions concerning typical problem solving behaviour in the context of a particular task and to provide an indication of the sorts of errors that problem solvers are likely to make in that task. Secondly, schema theory can provide an account of the way in which knowledge representation may change with increasing expertise. A description of this developmental process not only contributes to a theoretical understanding of the evolution of complex skills, but can also pose implications for instructional practice and for system design.

This chapter explores the relationship between the evolution of expertise and schema development and organisation in the context of a programming task. The chapter presents an empirical study which provides evidence for changes in the structure of programming knowledge with increasing expertise.

10.3 Schema development and expertise

Previous work concerned with programming and other complex problem solving tasks suggests a number of different ways of accounting for the development of expertise and its relationship to knowledge structure and organisation. These different views can be characterised briefly as follows:

Experts posses *more* schemata than novices

This is the position adopted by Soloway and Ehrlich (1984) in their analysis of the development of programming expertise. They introduce the notion of the programming plan to describe generic stereotypical knowledge structures or schemata that guide problem solving behaviour in programming. Soloway and Ehrlich claim that expert programmers possess this kind of programming knowledge while novices typically do not. According to this view, expertise develops through a gradual process of plan/schema acquisition and this process would appear to constitute the basic mode of learning in Soloway and Ehrlich's model. As a corollary, it has been suggested that one means of facilitating the development of expertise is to teach such plans explicitly to beginning programmers (Bonar and Cunningham, 1988).

Expertise and schemata *content*

Another way of accounting for differences in problem solving expertise is to suggest that experts have more complete schemata than novices (Heller and Riff, 1984; Larkin, 1983;1985). For instance, work by Chi, Feltovich and Glaser (1981) concerned with physics problem solving behaviour, suggests that while novices and experts may posses similar kinds of domain specific schemata, experts are able to access extra knowledge in comparison to novices in terms of elaborated schemata that represent general physical principles. Novices, by contrast, appear to represent only the 'surface features' of a problem and their problem solving behaviour is guided by keywords present in the problem description and by stated domain objects. Weiser and Shertz (1983) have replicated this finding for expert and novice computer programmers, thus suggesting that schemata content is an important general characteristic of competence in the programming domain.

Experts *structure* schemata differently

The studies reported so far in this thesis suggest a rather different view of the relationship between expertise and the development of knowledge representation. These studies suggest that experts may posses a greater number of program specific schemata than novices. This accords with Soloway and Ehrlich's characterisation of the development of programming expertise. However, as we noted in chapter 7, intermediates appear to be able to access the same range of schemata as experts. This is evidenced by the fact that intermediates can detect violations to plan or schemata structure as proficiently as experts. This finding would not be predicted by a view of programming expertise which places emphasis on the acquisition of program specific schemata such as that proposed by Soloway and Ehrlich. Rather, one would expect intermediate performance in response to schemata or plan violation to be better than novice performance and rather worse than expert performance. If one assumes that schema acquisition is continuous and correlated with developing expertise, as suggested by Soloway and Ehrlich, then these kinds of abrupt discontinuities in performance would not be predicted.

The model of knowledge restructuring presented in this thesis suggests that skill differences in programming may be related to the way in which programmers structure their knowledge about the programming activity. In particular, programmers who have attained high levels of skill appear to be able to access program schemata in which certain salient program structures have achieved prominence. According to this view, expertise is seen to be related in part to the development of hierarchically structured schemata, rather than to a relatively simple process of schemata acquisition.

Following Rist (1989) it is suggested that programs can be represented at different levels of abstraction. Central to Rist's elaboration of schema development in programming is the notion of the 'focal line'. According to Rist, a focal line directly encodes the goal represented by a particular program schemata or plan.

For instance, a plan or schemata concerned with calculating a running total will associated with the focal line 'count:=count + 1'. The complete plan will also consist of an initialisation component and some means of reading and outputting data values. Hence, one might conceive of focal lines as corresponding to the highest level of schemata abstraction. The other components of a particular schemata (i.e. non-focal lines) simply serve to extend this plan focus. This view of the programming activity was presented in the previous chapter, and the empirical data reported there appears to provide some support for this view.

If the development of expertise involves the restructuring of programming knowledge such that certain salient structures achieve prominence, then one might expect this to be reflected in the time taken to respond to the presence or absence of certain forms of stimuli. For example, if focal lines are taken to represent salient structures, then we might expect that when programmers are presented with focal lines and asked to state whether they were contained within a previously presented program which they were asked to memorise, they will respond faster than when presented with lines representing non-focal structures.

Moreover, since this restructuring process is hypothesised to be related to skill development, then we should expect to find differences between novice and expert responses in the context of this task. In particular, in the case of novices we have hypothesised that schema structures remain undifferentiated, hence we may predict that response times to focal and non-focal lines will be broadly equivalent. In contrast, experts will respond to focal lines more quickly than non-focal lines.

In the case of errors we might expect a slightly different result. For example, one major claim of this thesis is that while experts and intermediates may be able to access programming knowledge which has similar content, this knowledge is structured differently in the case of experts, and that this differential structuring leads to their enhanced performance in the context of certain tasks. If this is the case, while we might expect reaction times to differ when comparing intermediate and expert performance, their error rates should be broadly equivalent. These

specific hypotheses are evaluated in the experiment described below, in which programmers of different skill levels were asked to memorise a program and were then presented with probe items corresponding to focal or non-focal lines. A number of these probe items were derived from the programs the subjects were originally asked to memorise, while others were derived from other, similar, programs.

10. 4 An experimental study of schema development in programming

10.4.1 Participants and procedure

The experimental study reported in this chapter employs a program memorisation and recognition task to explore the form and content of knowledge representation for programmers of different skill levels. Twenty Four subjects participated in this study. These subjects were assigned to three groups of equal size according to their level of programming expertise.The novice group consisted of first year undergraduate students, all of whom had completed a short introductory course in Pascal. The intermediate group was composed of second year undergraduate students. All members of this group had undertaken an intensive 6 month Pascal course during their first year of study and all had subsequently used this language extensively in project work. The third group consisted of expert Pascal programmers who were either teachers of Pascal or were professional programmers.

Participants were presented with a number of short Pascal programs which they were asked to memorise. These programs were drawn from the collection of programs generated by subjects taking part in the experiment reported in the previous chapter. The participants were then presented with a probe item (focal or non-focal line) and were asked to state whether this probe item was present in the original program.

Half of the probe items were derived from the programs the participants were asked to memorise, while the remaining half were generated from other similar Pascal programs. A measure of reaction time from the presentation of the probe item to the subjects' response was collected together with information relating to the frequency of errors. In this context an error could be said to occur when a subject affirmed that a probe item was present in the original program when in fact it was absent or vice versa.The results of this study are represented graphically in figures 10.1 and 10.2.

The programs that were presented to subjects were classified according to the scheme outlined in the previous chapter. A focal line was defined as the line of code that directly implements the current goal of a particular programming plan. Three independent raters identified the plan structures contained in each program and then indicated the focal line associated with each of the plans that they had identified. It should be noted that there was a high level of agreement between the raters in terms of both the plans that were identified and their associated focal lines.

10.4.2 Results and discussion

This study suggests that the transition from lower to higher levels of skill in programming does not follow a continuous developmental path as might be suggested by certain models of programming expertise. In particular, if expertise depends upon the acquisition of program schemata or plans then one would expect to find a roughly linear relationship between increasing expertise, the number of errors made and the subjects' reaction time in response to probe items.

Hence, if experts possess a greater number and range of schemata than intermediates or novices then one might hypothesise that experts would be able to find a match to a probe item with greater speed and accuracy than both novices or intermediates. Additionally, intermediates should perform with greater accuracy

and speed than novices. The results of the present study, however, do not provide support for such a view. Rather, there appear to be clear discontinuities between subjects' response times and the number of errors made when comparing the performance of novices, intermediates and experts.

The results of this study also provide support for the view that knowledge structures change with increasing expertise. Hence, experts respond to the presence of focal lines with greater speed than both intermediates ($t = 2.83$, 14 *df*, $p<0.02$) and novices ($t = 5.72$, 14 *df*, $p<0.001$). However, there is no significant difference between experimental groups in terms of their reaction to non-focal lines. This suggests that in the case of the expert group, focal lines are represented with greater saliency than non-focal lines. However, subjects in both the novice and intermediate groups do not appear to posses schemata which differentiate between focal and non-focal lines. The error data indicates that intermediate and expert groups identify both focal and non-focal lines with greater accuracy than novices ([focal] $t_{intermediate}= 3.42$, 14 *df*, $p<0.01$; $t_{expert} = 8.63$, 14 *df*, $p<0.001$) ([non-focal] $t_{intermediate}= 5.83$, 14 *df*, $p<0.001$; $t_{expert} = 7.93$, 14 *df*, $p<0.001$).

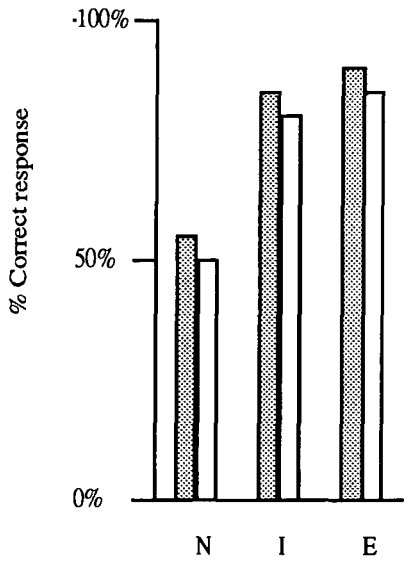


Figure 10.1

% Correct response to probe item for Novice (N), Intermediate (I) and Expert (E) programmers.

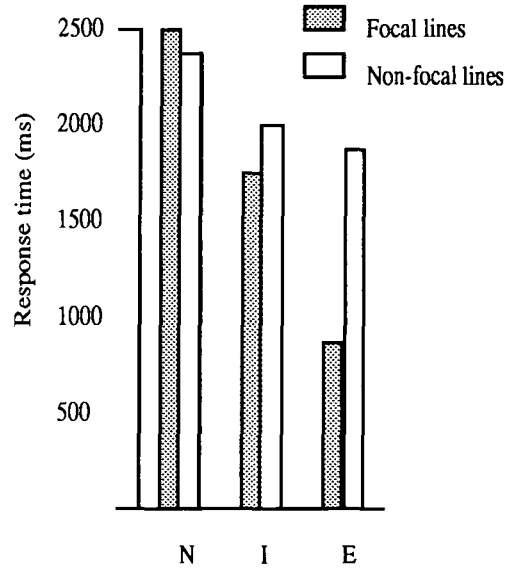


Figure 10.2

Mean response time (ms) to probe item by Novice (N), Intermediate (I) and Expert (E) programmers.

Conversely, both the intermediate and expert groups identify focal lines with about the same accuracy as non-focal lines ($t_{intermediate} = 1.23$, NS; $t_{expert} = 1.01$, NS). This finding supports the view that intermediates and experts are able to access representations of programming knowledge which are composed of similar components. However, as suggested by the reaction time data, these representations appear to be differentially structured. In the case of the novice group, performance is only slightly better than chance ($t = 1.14$, NS). Hence, novice performance does not appear to be mediated by the possession of complete schematic representational structures.

In summary, these results suggest that both intermediate and expert programmers appear to be able to access abstract schemata based structures which represent stereotypical programming knowledge. The error data for these groups implies that these schemata have similar content since they give rise to equivalent patterns of error behaviour for both groups. However, in the case of the expert group, reaction time performance in response to the presentation of a probe item is enhanced. This effect is hypothesised to be related to the existence of a structural asymmetry in the expert programmers schematic representation of programming knowledge.

10.5 Conclusions

The results of the experiment reported in this chapter suggest that knowledge structures may develop via a restructuring process rather than through a process of knowledge accretion which simply involves developing a larger repertoire of plans. This view is similar to work by Lewis (1981) which suggests that algebra skill depends partly upon the creation of single variables to replace complex mathematical expressions. This restructuring or 'information hiding' process may allow more complex problems to be accommodated within the limited capacity of working memory thus facilitating problem solving.

Previous accounts of programming expertise have tended to emphasise the development of extensive repertoires of programming knowledge rather than focus upon issues relating to knowledge restructuring processes. This is exemplified in Soloway's work on plans, where it is suggested that expertise simply involves building a more extensive collection of programming plans together with the rules which goven their use. However, as we have seen, the possession of plans per se cannot be used to diffrentiate certain groups of programmers, especially at higher levels of skill; namely at the point of transition between intermediates and experts. The work reported in this chapter suggests that a more cogent account of skill

development might combine knowledge accretion with a restructuring process whereby, once a sufficient library of plans has been constructed, fine-tuning or restructuring takes place. This view is similar to Rumelhart and Norman's account of skill devlopment which places central emphasis upon restructuring mechanisms founded upon the accretion of a large body accumulated domain knowledege (Rumelhart and Norman, 1978).

The model outlined above also has clear parallels with work on program beacons reported by Wiedenbeck (1986). Wiedenbeck found that expert programmers recall key lines (beacons) in a program much better than they recall non-beacon lines. However, the recall of non-beacon lines does not differ significantly when comparing novice and expert performance. This suggests that there are key features of programs that play a focal role in program understanding, and from previous work, it seems likely that these features guide the comprehension (Brooks, 1983) of programs. Moreover, in chapter 8 it was suggested that these focal structures may also play a significant role in program generation. The present study advances this analysis by providing a means of defining salient knowledge structures in programming in a less intuitive manner than has previously been possible. In addition, the focal line analysis presented here extends existing plan theory of programming by suggesting a model that can account for plan restructuring and its relationship to skill development.

The articulation of this model suggests that previous accounts of skill development in programming may be too simplistic to provide an adequate account of the development of expertise in this domain. In particular, it has been demonstrated that the existence of plans per se cannot account for the development of skill in programming. Rather, what appears to be more important is the way in which the organisation of individual schemata changes with increasing expertise.

Chapter 11. Expertise in programming: The role of working memory and display-based problem solving.

11.1 Introduction

The work reported in this chapter represents an attempt to explore the role of working memory in programming skill. Two experiments are reported in this chapter which demonstrate that the development of expertise in programming does not appear to depend upon an increase in working memory capacity or availability. This finding is unexpected given the importance placed upon working memory capacity in other theoretical accounts of the development of complex skills. These experiments provide evidence for an alternative view which suggests that expertise in programming may be dependent upon the development of strategies for efficiently utilising external displays for the purpose of recording intermediate states and partially formed solution steps.

In this context, it appears that novices rely extensively upon working memory to generate as much of a solution as possible before transferring it to an external source. In contrast, experts engage in problem solving behaviour which is characterised by the extensive use of an external display as an information repository. This gives rise to a pattern of generation behaviour which manifests itself in terms of a closely linked cycle of code generation and evaluation activities. One of the most striking results of this work is that when experts are unable to use an external display to support facets of this activity then their performance deteriorates to the level typically exhibited by novices. These results are discussed in terms of a framework which emphasises the role of display-based problem solving and its contribution to strategy development. Finally, the implications of this study are discussed in terms of its ability to account for performance in other complex problem solving domains.

As we have seen in previous sections, until recently the prevailing focus of study in the psychology of programming has been concerned with the organisation of knowledge in long term memory and the role of certain forms of conceptual organisation in program comprehension and the development of expertise.However,

more recent studies have moved away from the consideration of static knowledge structures towards a view which emphasises the fluid and dynamic role played by programming knowledge in both the generation of code (Green, Bellamy and Parker, 1987) and in program design (Rist, 1986; 1989). Other studies have focused upon the generative and evaluative aspects of the coding activity. For instance, in chapter 8 the cognitive processes involved in programming were characterised by studying the nonlinearities in program generation that are evident in naturalistic task contexts. In a somewhat similar vein, Gray and Anderson (1987) have used so-called 'change episodes' - key junctures in the coding process where programmers make some change to their code - to highlight the important role played by the evaluative or checking activities that are normally invoked during code generation.

These studies have extended our basic understanding of the strategies employed by problem-solvers in this relatively complex domain. However, they have tended to provide only rather general descriptive accounts of the cognitive processes which are thought to underpin the programming activity. Such studies have emphasised the effects of either different knowledge structures or of the salient notational features of the task language on the development and maintenance of particular types of strategy. However, by and large, they have ignored, or have addressed in only a cursory fashion, the nature of the basic cognitive mechanisms that may give rise such strategies. For instance, while it is true to say that such studies have explored the form of representation of programming knowledge in long term memory, and the effect this has on problem-solving strategy, they have typically only briefly addressed the role of other memory structures which may be closely implicated in the emergence of typical forms of programming behaviour.

The study reported here seeks to extend existing work into the cognitive aspects of programming by examining the role of one seemingly important cognitive mechanism - working memory - in the development of programming expertise. In addition, interest is directed towards understanding the way in which this limited resource might influence the nature of the problem-solving processes involved in the programming activity.

The role of working memory has been implicated in a number of previous studies of programming behaviour. However, as a mechanism that might be thought to strongly contribute to programming strategy, it has received little experimental attention.

This chapter will begin by briefly reviewing the existing work on programming that posits working memory as part of the framework within which the development of programming skill is discussed. Studies of working memory from other problem solving domains will be introduced in this section to provide a background to this discussion and to indicate any points of commonality or departure from those analyses suggested in recent studies of the programming activity. This provides a basis for the two experimental studies reported later in this chapter. The first of these studies looks at the effects of a straightforward articulatory suppression task on programming strategy and explores the role of working memory in the development of programming skill and its relationship to expertise.

The second study considers the relationship between working memory and the use of an external memory source (in this case a VDU screen). Here the text editor used to enter code was modified such that the order of program generation had to conform to the final text order of the program. That is, the screen editor only allowed the programmer to enter text vertically - top to bottom- and to move between adjacent lines. This can be contrasted to the more typical use of a screen editor where programmers can generate any part of a program in any particular order and can suspend the development of code in one place to direct attention to other code structures elsewhere, returning latter to fill in any remaining gaps.

A number of recent studies (Bellamy and Gilmore, 1990; Green et al, 1987 and see chapters 8 and 9) suggest that this is the way programmers typically develop their code. Conversely, if programmers have to commit themselves early to a particular course of action and are not able to effectively use an external medium to record their partially formed deliberations, then one might expect this to have both a detrimental effect on performance and to place extra load upon working memory. Previous studies which have explored the role of external memory sources have typically restricted their analysis to situations in which external sources are used to support long term memory (Intos-Peterson and Fournier, 1986; Schönflug, 1986a; 1986b).

The analysis reported here takes a different approach and considers the use of external memory support for short term and transient memory constructs. This approach also differs from other studies looking at the role of working memory in complex tasks (Hitch, 1978; Staszewski, 1988) in that typically these studies have required subjects to dispense with external aids even in tasks, such as simple addition and multiplication, where such aids appear to be used extensively under more normal circumstances.

The final section of this chapter is concerned with a detailed discussion of the implications of these experiments for our understanding of the role of working memory in programming and in other complex cognitive skills. In particular, attention is directed toward the role of working memory in the development of expertise, its role in complex planning tasks and mental simulation and in situations where the potential exists for the deployment of external memory sources which can be used to partially supplant internal sources.

11.2 The current status of working memory in studies of the programming activity

One of the most pervasive findings of recent research into the cognitive aspects of programming is that programs are not generated in a simple linear fashion - that is, in a strict left-to-right or first-to-last order (Bellamy and Gilmore, 1990; Green et al, 1987 and see chapters 8 and 9). Typically, programmers make many deviations from a strict linear development path, leaving gaps in the emerging program which are to be filled in later. Hence, the final text order of the program will rarely correspond to its generative order.

Existing models of programming suggest a number of possible reasons for the existence of such nonlinearities. For instance, Rist (1986 a and b;1989) claims that nonlinearaities in generation emerge as programmers build code structures around so-called 'focal lines'. The development of expertise in Rist's model is based upon the availability of knowledge and the notion of focal expansion. According to Rist, as expertise develops, plan schema are retrieved rather than created and the evolution of this process is reflected in the order of program generation. In terms of this model, novices are seen to generate programs in a bottom-up fashion expanding

outwards from the plan focus, while experts are able to simply retrieve plans in schema order, thus creating a pattern of top-down and forward program generation. This explains the differences observed in Rist's study between the strategies adopted by novices and by experts and the patterns of nonlinearities encountered.

Green et al (1987) have proposed a number of extensions to Rist's model. In the parsing/gnisrap model of coding they describe the process by which a skeletal plan is instantiated in a programming notation. In the context of Rist's model, focal lines will be instantiated first and then other structures will be built around them. The order of generation of the program is determined largely by the knowledge that is available to the programmer and nonlinearities will only occur when a programmer starts to interleave new plans into the existing structure. The parsing/gnisrap model builds upon Rist's ideas but introduces one significant extension.

Unlike Rist's model, the parsing/gnisrap framework introduces a working memory component into the analysis of coding behaviour. The parsing/gnisrap model has a severely limited working memory capacity which forces it to use an external medium (eg the VDU screen) when program fragments are completed or when working memory has become overloaded.

In this model programs are not built up internally and then output to an external media from start to finish with a generative order that reflects the final text order of the program. This form of generation would presumably imply an unrealistically extended (and sustained) working memory capacity. However, the working memory limitations suggested by the parsing/gnisrap model give rise to other cognitive costs. This is because programmers will need to frequently refer back to generated fragments and in some way recreate the original plan structure which may have only been partially implemented in code.

The parsing/gnisrap model, in its current conception, has primarily focused upon the factors that influence the program parsing process. Hence, interest is directed toward the way in which certain notational features of programming languages can obscure or reveal the functional role of components expressed in that notation (Gilmore, 1986; Gilmore and Green, 1988).

Bellamy and Gilmore (1990) examined the number of nonlinearities in terms of whether they occurred between plans (in the Soloway sense) or within plans. Using this technique they were able to compare predictions stemming from Rist's model with those implied by the parsing/gnisrap model. In Rist's model nonlinearities will only occur when a programmer starts to interleave a new plan into an existing structure. The parsing/gnisrap model suggests that nonlinearities can occur in other ways. For instance, minor parts may often be left out of plans and programmers will return later (in terms of the linear structure of the program) to insert these code fragments.

Hence, Rist's model suggests that there will be more between plan than within plan nonlinearities, while the parsing/gnisrap model predicts that generation will be broadly in plan order , but that within plan nonlinearites will occur because of working memory limitations. The results of the Bellamy and Gilmore study broadly support the predictions stemming from the parsing/gnisrap model and provide some evidence for the influence of features of the notation of particular languages in determining the nature of the parsing/gnisrap cycle.

11.3 External memory and display-based performance in problem solving

The parsing/gnisrap model displays several important commonalities with emerging models of planning and problem solving in other complex domains. In particular, this model stresses the role played by external memory sources as information repositories which can be used to record intermediate problem solving states. In a similar vein, work connected with the development of planning models in Artificial Intelligence has suggested that actions are often enacted before plans or problem solution sequences are complete (see chapter 2). In formulating this alternative account of planning, these models stress the inexorable link between the planning process and the execution of plans.

Models such as this differ significantly from the more classical accounts of planning that have been previously articulated in Artificial Intelligence in that an entire sequence of plans need not be worked out in advance. Rather, the effects of implementing partial plans can be tested against the planner's expectations and

information may then be sought from the external world in order to reduce the uncertainty that may be associated with the implementation of particular plans.

More recently, the role of external memory sources as repositories for search control knowledge and intermediate state information has gained prominence in a variety of problem solving models (Howes and Payne, 1990; Larkin, 1989; Payne, 1990; in press). For instance, Larkin (1989) has proposed a production system model of human 'display-based' problem solving (DiBS) where the system's working memory is divided into two kinds of elements: those reflecting the problem solver's internal goals and those representing features of external real world objects. As in other production system models, specific productions are executed when their preconditions are satisfied in working memory (Anderson, 1983). The associated actions of these productions then act in turn to modify the content of working memory. In terms of Larkin's model, these changes can arise in two ways - either as changes to the physical world or as changes to the problem solver's internal representation of goals and subgoals.

These accounts of problem solving and planning are clearly very similar in their general form to emerging accounts of problem solving and planning in the programming domain. For instance, a central feature of the parsing/gnisrap model is the idea that code generation involves two fundamental psychological processes; one in which the external structure (program code) is created from the internal (cognitive) structure that represents the problem requirements and an inverse process by which this internal structure is recreated when necessary from the external structure.

Similarly, Gray and Anderson (1987) stress the importance of the evaluation episodes that are frequently seen to occur during code generation. The existence of these evaluative activities presumably implies that programmers are able to extract relevant information from the code that they have already generated in order to inform their subsequent problem solving activity. This will necessitate re-parsing the code and then matching it with an internal representation of plans and goals. Hence, in terms of both of these models, the external display becomes a central repository for intermediate state information when working memory is loaded. The parsing/gnisrap model implies that code will not be generated in a linear fashion, since code fragments will be externalised as soon as working memory is loaded or

when the programmer arbitrarily reprioritises some task that demands a different series of activities than are required by the current goal.

11.4 Expertise and skilled memory theory

Although the parsing/gnisrap model relies extensively upon the notion of working memory in order to explain the evident nonlinearites in program generation, it fails to address several key issues in relation to the role of working memory that have been raised elsewhere. One issue of particular importance is the relationship between working memory and the development of expertise. The emphasis of the parsing/gnisrap model directs our attention towards the effects of language structure and notation on the parsing/gnisrap cycle. However the effects of skill development on strategy are not considered.

In chapter 8 a study was reported looking at the nature of the nonlinearities found to exist in a program generation task for programmers of different skill levels. One finding to emerge from this work is that experts perform a significantly greater number of between plan jumps than novices and that novices correspondingly tend to perform more within plan jumps - that is, adopt a linear generation strategy. This result is unexpected for two reasons. On the one hand, the parsing/gnisrap model assumes that working memory is a fixed and limited capacity resource. Hence, if we assume that working memory is the only factor (excluding language features) to influence the parsing/gnisrap cycle, then expertise should not affect the number of between-plan nonlinearities that are evident in the generation task.

Conversely, if we consider working memory to be a more flexible resource with an extensible capacity that is related in part to skill development (Chase and Ericsson, 1982), then we should expect to find the obverse result. Moreover, the chunking, restructuring and compilation mechanisms that are central to many important production system models of skill development should give rise to a reduction in working memory load in the case of expert performance (Anderson, 1983; 1987; Newell and Rosenbloom, 1981; Laird, Newell and Rosenbloom, 1988). Here, it is assumed that the problem solving steps and intermediate states required to reach a solution will be reduced in number when productions are collapsed, compiled or

otherwise restructured. This would presumably reduce the requirement to concurrently maintain a large number of solution steps in working memory and consequently increase its availability.

If we assume that the re-parsing of some already generated output involves some cognitive cost, then one might expect the development of programming skill to be at least partly dependent upon a programmers ability to generate as much of the program internally before writing it to some external source, and therefore reducing the need to re-parse. However, the opposite appears to be the case. The results of the experiments reported in chapters 8 and 9 may suggest that skilled programmers make extensive use of an external memory source (i.e., a VDU screen) while novices tend to rely to a much greater extent upon the use of internal memory to develop as much of the solution as possible before transferring it to external memory. In addition, novices appear to rarely change their output once it is generated.

In light of the well documented relationship between working memory and the development of expertise that has been observed in other domains (Chase and Ericsson, see above) these findings are clearly rather anomalous. Given the cognitive costs that are involved in continually evaluating and modifying generated code, we require an explanation as to why skilled programmers rely so extensively on external rather than internal memory sources.

One reason for this phenomenon might be related to the way in which programming is taught - in particular to the effects of the teaching of program/software design, and their emphasis on functional decomposition and stepwise refinement. The effect of teaching programmers to decompose problems into manageable components, as suggested by these methods, may have a strong influence on their adoption of external sources - for instance, to record partially formed design decisions etc.- in preference to a reliance on internal memory. Unfortunately, most existing studies of working memory in the context of skill development have tended to artificially restrict the task under consideration by disallowing the use of external sources. Hence, the ecological validity of these studies may be limited to the rather unusual skills that are considered. One important component of skill development in a number of domains, including computer programming, seems to be the related to the

ability to efficiently use external memory sources - and this component has not been extensively considered in previous studies of problem solving in complex domains.

Another factor that might give rise to a reliance on external sources is related to the question of whether the working memory system has a semantic component. That is, whether such a system can deal simultaneously with information derived from different semantic categories or levels of abstraction. When developing a program, information from a variety of sources needs to be assimilated, ranging from abstract information about high-level goals and plans to very low-level information relating to syntactic conventions etc. Indeed, the use of multiple forms of information is not only evident in programming. For instance, Kaplan and Simon (1990) have observed that problem solvers may need to search a variety of different problem spaces. However, in Kaplan and Simon's framework it appears that problem solvers are only able to access one problem space at a time. This is also evident in other generic problem solving models - e. g., SOAR (Laird, Newell and Rosenbloom, 1987).

It has been argued that programming demands the assimilation of information from a range of problem space representations (Pennington, 1987a; Pennington and Grabowski, 1990) and this assimilation of representations may need to be coordinated and integrated simultaneously. However, this kind of coordination and integration of information has been shown in many studies to place a significant load on working memory (Elio, 1986; Logan, 1985; Schneider and Detweiler, 1987). Hence, it may be the case that continually switching between these different abstraction levels incurs too great a cognitive cost and that a more efficient strategy may involve developing a solution at one level of abstraction before moving on to a lower level. This may give rise to the observation that expert programmers appear to develop their code from focal structures, building the rest of the code around these fragments and using the external display as repository for intermediate solution steps. Conceiving of the programming task in this way allows us to consider the interactions between display-based problem solving and partial planning, and may provide a more coherent understanding of the complex problem solving activities that take place in programming and other tasks.

The nonlinearities observed in the context of skilled programming performance, where high level components (focal lines) are generated first before other lower level structures is amenable to this form of explanation. Of course, this form of top-down development has been reported extensively in studies of the program design activity (Adelson and Soloway, 1985; Guindon et al, 1987; Jeffries et al, 1981), but it is usually accounted for in terms of the organisation of conceptual knowledge in long term memory rather than in terms of more transient and short term memory structures. As little seems to known about the possible semantic component of working memory, an account of programming strategy based upon this kind of working memory limitation seems to be equally valid.

Deviations from a top-down step-wise refinement approach to problem solving may be caused simply by working memory capacity limitations. Hence, problem solvers may simply forget details of their current goal and begin to pursue goals at other levels of abstraction. However, an alternative explanation might be to suggest that problem solvers find it more difficult to assimilate information in working memory when it is derived from a number of different abstraction levels. Anderson (1983) suggests this possibility when discussing proposed opportunistic control structures for cognition (Hayes-Roth and Hayes-Roth, 1979) where information from a variety of different hierarchical levels is assimilated in working memory during problem solving activities. Anderson claims that opportunistic control of this kind "causes problems because skipping among its many planes and levels makes unrealistic demands on working memory" (p 130). Hence, we may suggest that in cases such as this the use of an external display becomes paramount. Moreover, the typical form of programming strategy exhibited by experts may simply be a manifestation of the use of such external resources and the fact that this necessitates many interlinked planning and evaluation cycles.

11.5 Experimental Studies

The present chapter seeks to explore a number of the issues outlined above in terms of the two experimental studies reported in this section. While previous research provides a strong indication of the role of working memory in programming, there exists no empirical research which has addressed its role more directly. The standard

paradigm of working memory research is adopted for the first experiment. Here, subjects were requested to carry out a straightforward articulatory suppression task while engaged in a concurrent program generation activity. Here, the number of within and between plan nonlinearities were recorded as were the number of errors remaining in the program on task completion.

This experiment addresses a number of specific hypotheses. Firstly, if working memory limitations cause programmers to make use of an external medium, as suggested by Green et al, then the act of loading the working memory system through a concurrent task should give rise to an increase in nonlinearities. Given the effort required to use an external medium, in terms of the number of times a programmer must engage in the parsing/gnisrap cycle, one would expect experienced programmers to rely more extensively upon internal sources. Additional support for this hypotheses also arises from a range of studies which suggest that skill development is accompanied by an increase in working memory capacity or availability (Chase and Simon, 1982; Staszewski, 1988) .

However, the results presented in chapter 9 suggest a rather different picture and indeed give rise to an opposing hypothesis, i. e., that skilled programmers make less use of internal sources than do novices and tend to rely much more extensively upon using an external medium to record partial code fragments as they are generated. Hence, when working memory capacity is restricted this should give rise to a greater number of nonlinearites in the context of novice behaviour and only a small decrement in nonlinearites for experts. The first experiment attempts to address these specific hypotheses by recording the nonlnearities in program generation for novice and expert programmers under normal task conditions and in the situation where working memory is loaded via a concurrent task.

The second experiment considers the role of working memory from a rather different perspective. Here interest is directed towards the way in which restricting the use of an external medium affects programming performance. In terms of the above analysis, if programmers are not able to correct already generated code at later stages in the coding process, then this should have an effect on their performance. Programmers can use a variety of different media to record a program as it is generated and the nature of this media is likely to affect the programming process

itself. For instance, pen and paper provides great flexability in terms of changing an existing output - programmers can simply correct errors as they become apparent. In addition, pen and paper can provide a means of indicating commitment to some output - for instance, using light rather than heavy strokes (Green, 1989).

Screen editors can provide similar flexibility, but require users to engage in additional effort in terms of formulating and issuing editing and search commands etc. An example of a highly restrictive environment is that of a simple line-editor where the complexity of editing and search is exacerbated, since users often need to explicitly address a particular line of text to carry out some editing operation. One can envisage even more restrictive environments where it is not possible at all to decouple the order of program generation from its final text order. For instance, Green (1989) suggests that the failure of a prototype Pascal speech-input system was caused largely by the fact that users were required to dictate the program in final text order, rather than in the order in which they might normally generate it.

The second experiment required subjects to create a program using a full-screen editor that provided no opportunity for the subject to revise already generated input or to insert new material into existing text. The use of such an editor will clearly place a significant load upon a subjects working memory capacity since they will be required to internally generate as much of the program as possible before externalising it. Anderson and Jeffries (1985) have demonstrated that many errors in programming arise when there is a loss of information in the working memory representation of a problem. Hence, by placing emphasis upon the use of working memory it should be possible to induce error prone behaviour that parallels that evident when working memory is loaded in other ways, for instance via articulartory suppression.

In terms of the previous discussion, we hypothesise that experts would perform worse than novices when the device used to create the program is restricted in such a way as to make retrospective changes impossible. This is based upon the assumption outlined above which suggests that experts make more extensive use of external sources to record partial code fragments that are then later elaborated and extended through a closely linked cycle of generative and evaluative activities.

Conversely, it has been suggested that novices will tend to rely to a greater extent upon generating as much of the program as possible internally before writing it to an external source. It is clear that these strategic differences will be supported to a greater or a lesser extent by the device used to create the program. Hence, for expert programmers, it is hypothesised that restricting the device will cause them to revert to a novice strategy, since they will then be unable to use the external display in the normal way. This, in turn leads to the hypothesis that novice and expert error rates will be similar in the restricted device condition, reflecting a decrement in performance by the expert group.

Establishing support for this hypothesis would have a number of important implications. Firstly, it would suggest that the development of expertise may not be based simply upon the acquisition of knowledge about a particular domain. If this were the case, we would expect experts to perform better than novices no matter what constraints were imposed by the task environment. Secondly, it would indicate that increased working memory availability, perhaps arising through some mechanism such as chunking, does not necessarily lead to better performance.

If increased working memory availability is correlated with expertise, then experts should perform better that novices in situations where they are required to rely almost exclusively upon internal sources. If this is not the case, then it may become necessary to review the central status of working memory in theories dealing with the development of complex skills. One possible alternative explanation is to argue that experts have developed particular strategies for dealing with task complexity that involve close interaction with external information repositories to record partial solution fragments as they are generated. If novices have not developed such strategies, then it is unlikely that their performance would be affected significantly by restricting the task environment.

This analysis can be further extended by attempting to classify the errors in the programs generated by subjects. For example, a scheme devised by Gilmore (1986b) suggests four main categories of error:

1. Surface level errors caused mainly by typing and syntactic slips: For example, confusion between < and >, missing or misplaced quotes etc.

2. Control-Flow errors: For example, missing or spurious else statements, split loops etc.

3. Plan-Structure errors: Including, guard test on wrong variable, update wrong variable etc.

4. Interaction errors: A class of errors occurring at the point where structures of different types interact: For example, a missing 'Read' in the main loop, initialisations within the main loop.

Clearly some of these errors will be knowledge-based (specifically, plan-structure errors) while others will be strongly dependent upon working memory limitations. For example, both control-flow and interaction errors, since they depend upon the establishment of referential links and dependencies between code structures, are likely to be affected by working memory constraints. Hence, in terms of the first experiment, we might expect that both control-flow and interaction errors will predominate in novice solutions in the situation where working memory availability is reduced. In the case of experts, it is argued that the interactions and interdependencies between code structures will be evaluated in the context of using an external memory source. That is, by reparsing existing code fragments in order to reconcile them with the code the programmer is currently working on. As a consequence, that the act of loading working memory will therefore not affect the occurrence of these types of error.

In the case of the second experiment we would expect the converse. If experts are not able to use the external display to aid problem solving in the manner predicted, then it might be hypothesised that interaction and control-flow errors will predominate in the condition where use of the device is restricted such that retrospective changes cannot be made. It might also be predicted that this experimental manipulation will not affect the occurrence of plan-structure errors since these are hypothesised to be knowledge-based rather than strategy-based.

11.5.1 Experiment 1. The effects of articulatory suppression on programming strategy and errors

Method

Subjects

Twenty subjects participated in this experiment. One group of ten subjects consisted of professional programmers who were employed in commercial or industrial contexts. All the subjects in this group used Pascal on a daily basis and all had substantial formal training in the use of this language. Members of this group were classified as experts. A second group of ten subjects consisted of second year undergraduate students all of whom had been formally instructed in Pascal syntax and language use during the first year of their course. None of these subjects had used Pascal before, however some had prior experience of other languages (predominantly BASIC and COBOL). Members of this group were classified as novices.

Materials and procedure

A variety of suppression tasks were explored initially, but the more complex tasks tended to disrupt performance to such an extent that a very simple articulatory suppression task was adopted. This involved asking the subjects to repeat a string of five auditorily presented random digits. Subjects engaged in this task until they had completed the experimental session. Hitch and Baddeley (1976) found that a concurrent task similar to this significantly affected verbal reasoning performance in the context of an experimental task which involved fairly simple logical deductions. Hence, it is reasonable to assume that this concurrent task would also affect performance in a more complex activity such as programming.

The experimenter was present throughout the session in order to ensure that this concurrent task was performed, and intervened only when the subject paused for more than 5 sec. The subjects were requested to generate a fairly simple program from a natural language specification. This specification required the subjects to

produce a program that could read a series of input values, calculate a running total, output an average value and stop given a specific terminating condition (see chapter 6 for a full description of this specification). This specification was derived from Johnson and Soloway (1985) and was chosen because it has formed the basis for many empirical studies of programmers. Hence, it was assumed that the resulting programs could be more easily analysed in terms of their constituent structures (specifically, plan structures) and in terms of any errors remaining in the program on completion of the task.

The subjects were allowed to study the specification for 2 mins. and were then asked to generate a program corresponding to this specification while engaged in the concurrent suppression task. The subjects were given 15 mins. to complete this task, and were asked to work as quickly and accurately as possible. The subjects typed their solutions onto a familiar full-screen text editor.

A non-invasive keystroke recording device was used to monitor the subject's behaviour. This device provides facilities for replaying keystrokes via a host machine. This replay was analysed to provide some indication of the temporal sequence in which programs were generated by subjects. In addition, three independent raters were asked to analyse all the resulting program transcripts for the presence of plan structures and for errors. Within and between-plan jumps were defined as follows: Within plan jumps were classified as movements between a particular line of the program text to another line which formed part of the same plan structure. Between plan jumps were defined as movements from the current line to lines within different plan structures. These protocols applied only to situations where the jumps was followed by an editing action (i.e., to insert, append or change some text). Finally, the errors were classified according to the scheme described above. That is, into surface, control-flow, plan or interaction errors.

Design

The experiment was a two-factor design, with the following independent variables:

1. Articulatory suppression/No suppression

2. Level of expertise - Novice/Expert

There were two dependent variables:

1. The number of Between/Within plan jumps during program generation
2. The number of errors remaining in the final program

Results

Plan-jumps

The results of this experiment are shown graphically in figures 11.1, 11.2 and 11.3. Figure 11.1 shows the number of within and between-plan jumps performed by novice and expert programmers in the two experimental conditions. These results were analysed using a three-way analysis of variance.

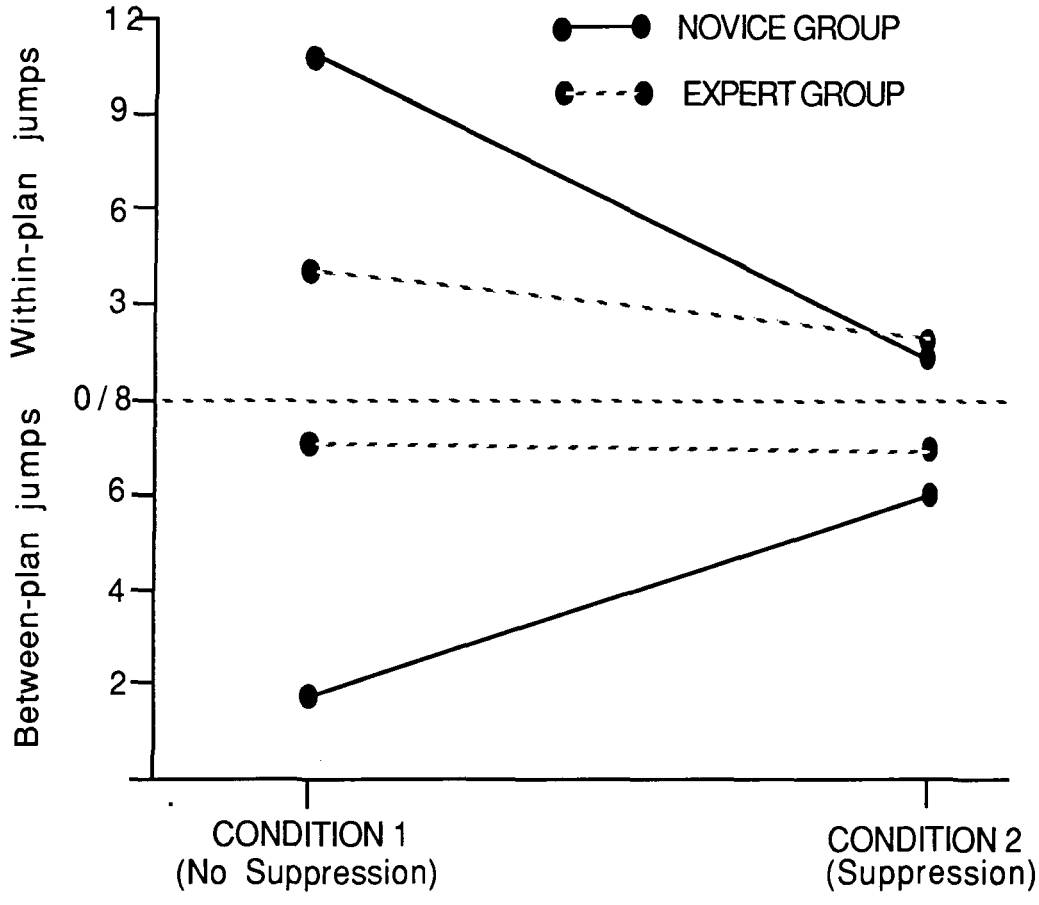


Figure 11.1 Number of within and between-plan jumps performed by novice and expert groups in the two experimental conditions.

This analysis revealed main effects of suppression ($F_{1,72} = 8.47$, $p<0.01$) and expertise ($F_{1,72} = 12.56$, $p<0.01$) on jump-type and a more complex interaction between suppression and expertise ($F_{1,54} = 4.73$, $p<0.05$). A number of post-hoc comparisons were carried out using the Newman-Keules test with an adopted significance level of $p<0.01$. This procedure indicated that experts produced significantly more between plan jumps than novices in the non-suppression condition. Conversely, novices produced a greater number of within plan-jumps in

this condition. In the case of the suppression condition, there were no significant differences between both within and between-plan jumps for either novices or experts.

The results of comparisons across conditions indicated a significant difference between the number of both within and between-plan jumps for the novice group. No significant cross condition comparisons were evident in the case of the expert group.

Errors

Figure 11.2 shows the total mean number of errors remaining in the programs on task completion for novice and expert subjects in the two experimental conditions. This was analysed using a two-way analysis of variance. This analysis revealed a main effect of expertise ($F_{1,36} = 9.37$, $p<0.01$) and suppression ($F_{1,36} = 4.54$, $p<0.05$) and an interaction between these two factors ($F_{1,36} = 15.89$, $p<0.01$). Once again a number of post-hoc comparisons were carried out using the Newman-Keules test with an adopted significance level of $p<0.01$. This indicated a significant difference in error rates in the both experimental conditions when comparing the novice and expert groups. In addition, a significant difference between error rates across conditions was evident for the novice group. In the case of the expert group the same comparison proved not to be significant.

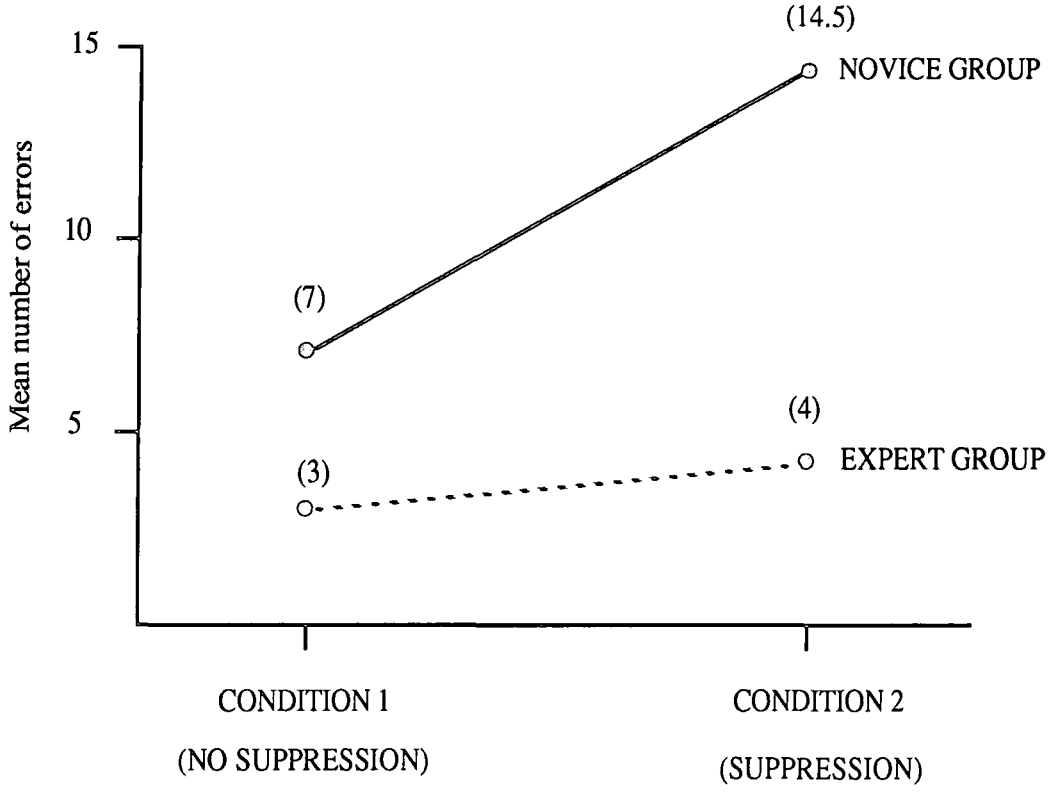


Figure 11.2 Mean number of errors in each of the experimental conditions for experiment 1.

Error classification analysis

Figure 11.3 represents the proportion of errors falling into each error classification. In the case of experts, there is a fairly even distribution of error types across the two experimental conditions. Indeed, further statistical analysis revealed no significant differences between error types both within and between conditions (multiple t-tests). In the case of the novice group, the distribution of error types is less straightforward. In the non-suppression condition, novices produced a significantly greater number of plan errors in comparison to the other categories (t-test). Moreover, the only significant difference between the novice and experts groups in this condition was the number of plan errors produced by the novice group (t-test).

In the second condition, the distribution of errors across classification types for expert subjects was again fairly even. No significant differences between any of the error classifications were evident (multiple t-tests). In the case of the novice group, significantly more control-flow and interaction errors were evident in comparison to the other two error classifications (t-test). Moreover, for the novice group,the number of plan errors occurring in the second condition was significantly less than in the first condition (t-test).

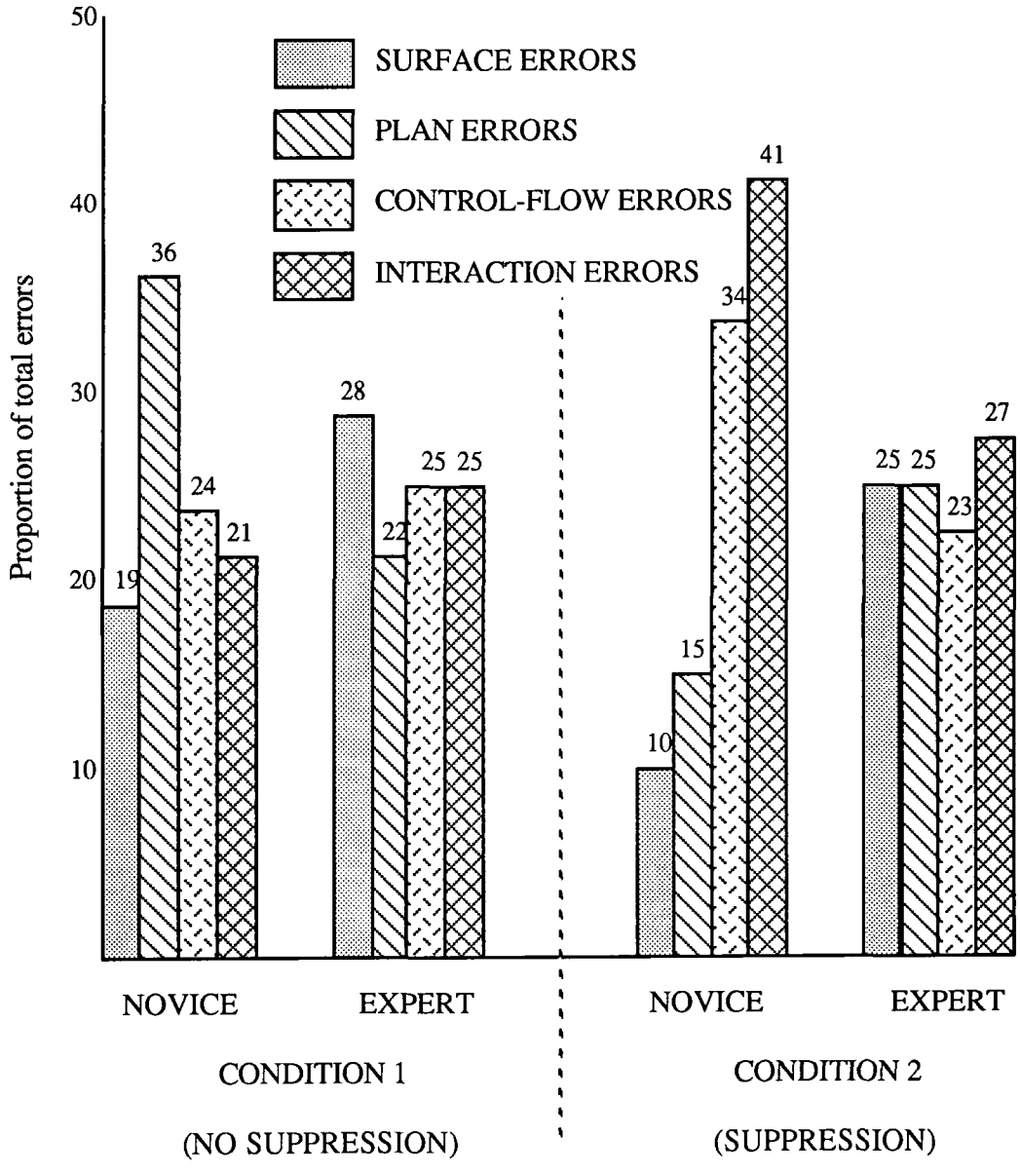


Figure 11.3 Proportion of errors in each error category in experiment 1.

Discussion

This experiment has clearly demonstrated that expert performance in programming tasks is not significantly affected by articulatory suppression. Hence, for experts the total number of errors remaining in the final program on task completion is not significantly different in the suppression condition in comparison to the non-suppression condition. Moreover, it appears that the nature of expert strategy is similarly unaffected by this experimental manipulation. Hence, the prevalence of between-plan jumps that is evident in the non-suppression condition for the expert group is not diminished in the suppression condition. Similarly, the occurrence of within-plan jumps does not differ significantly in the two experimental conditions.

Conversely, the novice group produced significantly more errors in the suppression condition when compared to the non-suppression condition. In addition, the nature of the coding strategy that they adopt is also affected. In particular, it appears that novice programmers revert from a linear generation strategy characterised by the prevalence of within-plan jumps, to a strategy more characteristic of experts. That is, to a strategy which reflects a greater number of between-plan jumps.

In section 11.5 it was stated that expert programmers appear to rely much more extensively than novices upon the use of external sources to record partial code fragments and that the act of loading working memory or of otherwise reducing its availability would not affect this process. Hence, in terms of that model, it was suggested that experts will tend to engage in very closely linked cycles of planning, subsequent code generation and evaluation. Since it is posited that this process relies very little upon the programmer's working memory capacity it is reasonable to expect that articulatory suppression would not affect the nature of performance in the context of this task. The results of this experiment provide support for this view.

Conversely, it has been argued that novice programmers place much greater reliance upon the internal development and simulation of code. In this case, it is claimed that reducing the availability of working memory will cause the novice programmer to revert to a strategy that necessitates more extensive use of an external media. Moreover, it is hypothesised that this change in strategy will be reflected in the number of discontinuities between discrete program structures. That is, in terms of

between-plan jumps. Once again, the results of this experiment provide some evidence for this account of programming behaviour.

Further support for this view is evident in the error data. In the non-suppression condition, novice subjects are clearly more error prone than experts. This finding is not unexpected. However, in the suppression condition, the error rate for the expert group changes little from this base line whereas the novice error rate more than doubles. This may indicate that when working memory is loaded novices must externalise information and that this constitutes a strategy which they find unnatural, thus leading to an increased error rate.

A more detailed analysis of these errors in terms of the classification scheme described in section 11.5 reveals a change in the nature of errors for novice subjects between the two experimental conditions. Hence, in the non-suppression condition, the novice group tend to make a greater number of plan errors, suggesting knowledge-based difficulties. Conversely, in the suppression condition a greater proportion of control-flow and interaction errors are evident. In terms of the analysis presented in previous sections, the preponderance of control-flow and interaction errors may simply reflect a difficulty in keeping track of the interdependences between various elements in the emerging program. When working memory availability is reduced it appears that novices experience some difficulty with these interdependencies. Moreover, unlike experts, it appears that novices cannot use the external display as an aid to memory to its full extent.

It could be argued that an alternative explanation for these findings is that experts simply have an extended working memory capacity and are not affected to the same extent as novices by a reduction in this capacity. Such an account would presumably have no difficulty predicting the results of the experiment reported above.
In order to assess the cogency of this alternative explanation the second experiment reported in this chapter adopts a rather different approach for exploring the relationship between working memory and the development of programming skill. In particular, if experts, for whatever reason, are able to extend their effective working memory capacity or increase its availability in other ways then restricting the task environment should not significantly affect their performance.

The second experiment reported in this chapter should be seen as complementary to the first. Whereas the first experiment involved an attempt to reduce the subjects' available working memory capacity, the second experiment has been designed so as to encourage subjects to rely extensively upon working memory. Hence, if experts have an extended working memory capacity then they should demonstrate performance equitable to that displayed in the first experiment. Moreover, if this extended capacity notion is correct, then clearly experts should perform better than novices even in the situation where the task environment is severely restricted as is the case in the second experiment.

11.5.2 Experiment 2. Effects of restricting the task environment

Method

Subjects

The same subjects participated in this experiment. However, the order in which they took part in each experiment was randomised.

Materials and procedure

Once again, the subjects were asked to produce a program corresponding to a brief specification written in English. This program (based upon the bank problem described in Johnson, 1988. See figure 11.4) involved processing three types of bank transactions. In this experiment, the nature of the task environment formed the basis for the two experimental conditions. In one condition, subjects were aksed to generate a program using a familar full-screen text editor. In the second condition subjects used a modified version of the same editor,which allowed cursor movement in only one direction. That is, from the top of the screen to the bottom. In addition, the text editor allowed only cursor movement between adjacent lines. Hence, once a subject had generated a line and pressed the return key, they were unable to then return to that line to perform any subsequent editing operations. The editor did, however, allow edits to the curent line being generated.

Write a Pascal program that can process three types of bank transaction: deposits, withdrawals, and a special transaction that indicates that no more transactions are to follow. Your program should begin by asking the user to input their account id and initial balance. The program should then prompt the user to input the following information:

1. the transaction type

2. if it is an END-Processing transaction the program should print out (a) the final balance of the users account, (b) the total number of transactions, and (c) the total number of each type of transaction. The program should then stop.

3. if it is a DEPOSIT or WITHDRAWAL, the program should ask for the amount of the transaction and then post it appropriately.

Figure 11.4 A specification of the 'bank problem'

The subjects first participated in a 5 min. familiarisation session, where the basic modifications to the editor were described. The subjects were then asked to attempt to generate a program from the specification. They were told to be as accurate as possible since they would be unable to change their input one they had pressed the return key at the end of each line. They were asked to check each line of their program before pressing the return key, in order to determine whether they were satisfied with their response. Subjects in both conditions were given 15 mins to complete this task. The subjects were randomly assigned to each of the experimental conditions.

Design

This experiment was a two-factor design with the following independent variables:

Environment - restricted/unrestricted

Level of expertise - Novice/Expert

In this case the dependent variable was the number of errors remaining in the final program.

Results

Errors

The results of this experiment are shown graphically in figures 11.5 and 11.6. Figure 11.5 shows the total mean number of errors produced by the expert and novice groups in the two experimental conditions. These data were analysed using a two-way analysis of variance with the following factors; Environment (restricted or unrestricted) and Level of expertise (Novice/Expert) This analysis revealed a main effect of Environment ($F_{1,36} = 5.74$, $p<0.05$), a main effect of Level of expertise ($F_{1,36} = 4.21$, $p<0.05$) and an interaction beween these two factors ($F_{1,36} = 9.76$, $p<0.01$). A number of post-hoc comparisions were carried out using the Newman-Keules test with an adopted significance level of $p<0.01$. This analysis revealed a significant difference between the number of errors produced by novices and experts in condition 1 (unrestricted environment). In condition 2 (restricted environment), this comparison did not prove significant. Comparisons across conditions revealed a significant difference in the number of errors produced by the expert group, but no difference in the case of the novice group.

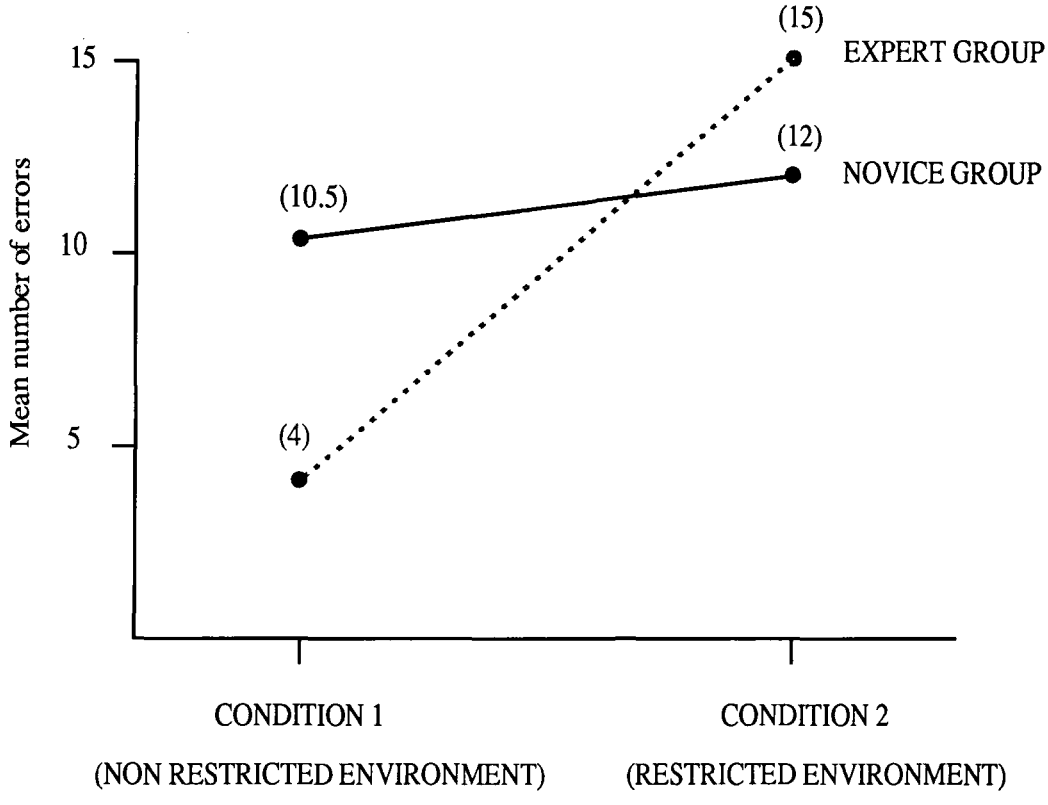


Figure 11.5 Mean number of errors in the two experimental conditions in experiment 1.

Error classification

The resulting program transcripts were analysed in terms of different error types according to the classification scheme described above. The results of this anlysis are shown in Figure 11.6. In the case of experts, there is a fairly even distribution of error types in the first experimental condition. Indeed, further statistical analysis revealed no significant differences between error types within this condition (multiple t-tests). In the case of the novice group, the distribution of error types in the first condition suggests a greater proportion of plan errors in comparison to the other categories (t-test).

In the second condition, the distribution of errors across classification types for expert subjects was rather more complicated. Here, experts produced a greater proportion of control-flow and interaction errors in comparion to the other error classifications (multiple t-tests). In addition, experts produced significantly more control-flow and interaction errors in comparison to the first condition. Experts also produced significantly more control-flow and interaction errors in comparison to the novice group in this condition. In the case of the novice group, there were no significant differences in terms of each error classification across the two conditions. As in the first condition, novices produced significantly more plan errors in the second condition.

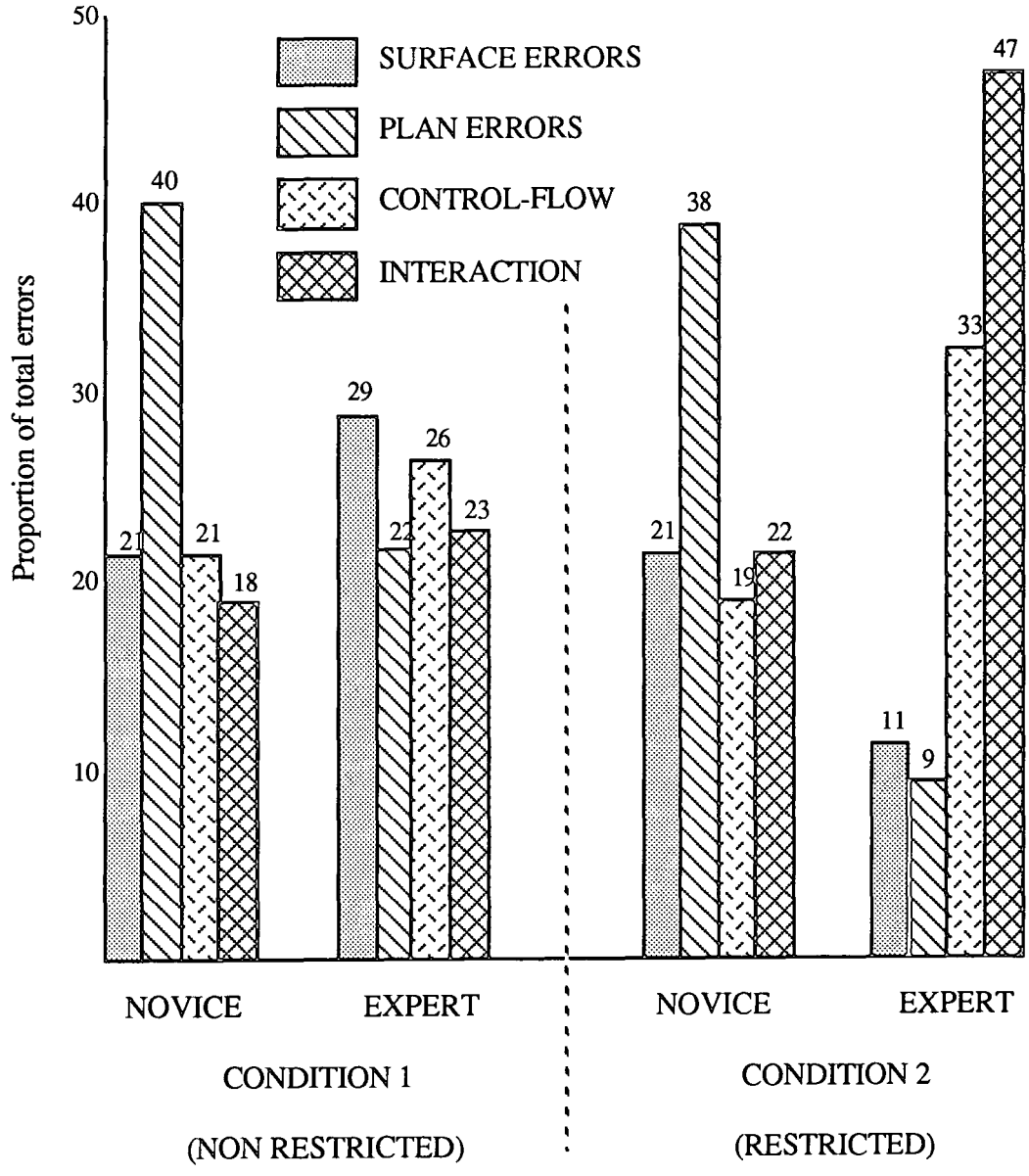


Figure 11.6 Proportion of errors in each error category in experiment 2

Discussion

These results provide a striking demonstration of the effects of restricting the task environment on problem solving performance. In previous sections it was argued that experts rely to a great extent upon using the external dispaly to record fragments of code that are then further elaborated at subsequent points during the generation process. This led to the hypothesis that if programmers were unable to return to previously generated fragments then they would be forced into a situation where they would have to rely extensively upon working memory to establish the various dependences and interactions between code structures. However, it appears that while novices are seemingly unaffected by changes to the task environment, experts not only perform worse than novices but also produce the kinds of errors that are indicative of an inability to internally construct links and interdependencies between code structures. Hence, these results reveal that experts produce more errors than novices in the restricted task environment. Moreover, experts produce a significantly greater number of control-flow and interaction errors in this second condition.

It was suggested in discussion of the first experiment that the results emerging from that study might reasonably be interprited as indicating that experts have an extended working memory capacity. However, if this were the case then the results of this second experiment would appear to be rather anomolus. In particular, if experts have an extended working memory capacity in comparison to novices, then we might expect that situations which cause experts to rely upon working memory would not give rise to such an extensive decrement in performance.

Moreover, there appears to be no reasonable explanation in terms of the increased working memory capacity assumption as to why experts produce many more control-flow and interaction errors in comparison to novices. Recall that in the first experiment it was the novice group that displayed a greater frequency of control-flow and interaction errors.

A more cogent explanation for these findings might simply involve suggesting that experts rely upon external sources and are not able to efficiently revert to a strategy that demands extensive reliance upon working memory. This analysis would

account for both sets of experimental findings. In the first experiment a reduction in working memory availability did not affect expert performance.

This could clearly be accounted for in two ways. On the one hand, it could be argued that experts simply have an extended working memory capacity. Conversely, we might claim that experts rely extensively upon external sources and find it difficult to adopt other alternative strategies. However, the second experiment appears to suggest that the first of these explanations is incorrect. In particular, if experts have an extended working memory capacity then we would expect them to perform better than novices in situations where a reliance upon working memory is necessitated. This appears not to be the case.

11.6 Conclusions

11.6.1 Working memory, display-based problem solving and the development of expertise.

These experiments have clearly demonstrated that the relationship between skill development in programming and working memory is not as predicted. Hence, it appears that experts rely significantly upon external sources to record code fragments as these are generated and then return later, in terms of the temporal sequence of program generation, to further elaborate and extend these fragments. It has been suggested that one of the major determinants of expertise in programming may be related to the adoption or the development of strategies that facilitate the efficient use of external sources. In contrast, novices appear to develop as much of the program internally before transfering it to an external source. In addition, they appear to rarely change their code once it has been generated

The reason why experts make such extensive use of an external medium is unclear. This externalsiation of information clearly has a high cost in terms of the reparsing or recomprehension of generated code that is implied. Hence, it might seem counterintuitive to suggest that problem solvers will tend to rely upon this kind of strategy rather than upon a strategy which involves the more extensive use of working memory. However, this explanation is consonant with existing work which

has implicated display-based recognition skills in theoretical analyses of complex problem solving (Larkin, 1989; Howes and Payne, 1990). The contribution of these analyses has been important, however they have neglected to consider the relationship between display use and expertise and the consequent effect that this may have upon the nature of problem solving strategies.

It is clear that a problem solver's ability to maintain prior states of control and to remember intermediate solution steps is severly limited (Greeno and Simon, 1974; Simon, 1975). In addition, Hitch (1978) has shown that working memory errors are reduced if intermediate results are utilised as soon as possible after they are produced. These factors may give rise to the development of strategies which cause the programmer to rely primarily upon the use of external memory sources.

The results of the experiments reported here question the cogency of accounts of skill development which place central emphasis upon the assumption that experts possess an extended working memory capacity or availability. In the case of expert programmers, it has been show that articulatory suppression affects neither programming strategy nor number of errors. This would be expected given the increased capacity assumption posited in previous models. However, the results of the second experiment would not be predicted on the basis of this assumption.

In particular, restricting the task environment such that the programmer must rely more extensively upon working memory should affect neither strategy nor errors to the extent that was apparent in this particular study. Since it is uncommon to see expert performance reduced to level exhibited by novices, it might therefore be assumed that restricting the environment in this way causes experts to revert from their adopted strategy to one more characteristic of novices.

Not withstanding this, there is a significant amount of empirical evidence for the increased capacity assumption, and as such it would be premature to reject it on the basis of the experiments reported here. However, these experiments suggest that display-based competence is an important factor in the development of expertise in programming. The extent to which the development of such a competence is relevant to other skills remains an important empirical question. Programming is clearly a complex skill, and this complexity appears to necessitate the use of an external

medium. In the context of other skills, it may be the case that an external medium is not available. Alternatively, other problem domains may not give rise to an extensive number of interacting solution steps as is clearly the case in programming. Hence, in other complex domains it is possible that there are very good reasons for relying upon working memory.

The present study may be limited in terms of its ability to account for problem solving strategies in other domains, although there are areas where this kind of analysis appears to be relevant, e.g., in writing and text composition, where multiple constraints and solution step interactions are inescapable (Flower and Hayes, 1980; Sharples and O'Malley, 1988). While the analysis presented here may not be appropriate to all problem solving domains, it is clear that, in terms of programming at least, this study poses a number of implications which may challenge the validity some of the claims made in previous analyses.

11.6.2 Working memory and the nature of errors

The work reported here has a number of implications for the way in which we might attempt to explain the occurrence and distribution of different types of error. In particular, it is clear that a certain classes of error can be attributed to working memory limitations and that such errors are not distributed at random. In terms of the error classification employed here, it appears that interaction and control flow errors predominate in situations where working memory availability is reduced. Previous work (Anderson and Jeffries, 1985) has suggested that errors arising from working memory failures will occur at random. More recently, Anderson (1989) has claimed that "working memory failures are slips at random and produce a wide range of different responses and would not concentrate at one place in the protocol" (p 350). However, the results of the present study suggest that working memory related errors may have a more systematic distribution, and that the type of errors one might expect to occur may to some extent be predictable.

11.6.3. The role of language features and the nature of task environments

Gilmore (1986a) has criticised Anderson and Jeffries' analysis of working memory errors from a rather different perspective. Gilmore suggests that their analysis "is very weak and its main impact lies in the new approach, rather than the detailed analysis. The main weakness is that language features do not seem to be considered relevant to the analysis. Anderson and Jeffries make no attempt to analyse the causes of processing overload ..." (p 528-529).

This criticism is pertinent to the present analysis since the nature of display-based problem solving in programming will be highly dependent upon features of the particular programming language considered. For example, Green (1990; 1991) suggests that some programming languages are "viscous" in that they are highly resistant to local modification. In terms of the analysis presented here, less viscous languages will provide better support for the kind of incremental problem-solving processes that are proposed. Hence, we might predict that programmers using different languages will make different kinds of error. In addition, since experts appear to employ an incremental strategy and novices a characteristically linear strategy, then it could be argued that some languages may be more suited to experts and others to novices.

The language features described above will affect the strategies employed in the generation of code. However, there are other language features which will affect its comprehension. Gilmore and Green (1988) suggest that some languages are "role-expressive" (for example, Pascal) in that they may contain a rich source of lexical cues which enable a programmer to distinguish more easily the constituent structures contained in a program written in that language. They contest that less role-expressive languages (for instance, Prolog) are lexically more amorphous and that such languages will not facilitate certain forms of comprehension.

More recently, claims have been made about object-oriented languages which may provide support for the present analysis. For example, Rosson and Alpert (1990) have claimed that such languages facilitate decomposition of the problem space by enabling programmers to develop encapsulated chunks of code whose internal operations are effectively isolated from other chunks. They go on to claim that this

form of decomposition will increase the amount of information that can be held in working memory since objects in the problem space can be held as separable chunks whose lower order implementation has no implications for other objects of interest. This arises because object-oriented languages enable a programmer to establish an abstract interface to a data structure which effectively hides the implementation details from the procedure using that data. All access to a data structure is effected via operations provided by the data structure's public interface. In object-oriented languages, the data contained within an object are private to that object and are accessible only via messages to the owning object (Micallef, 1988). Hence, the message interface can make useful information about an object available while hiding its implementation details (Goldberg and Robson, 1983).

Rosson and Alpert claim that this kind of encapsulation will reduce working memory load since the programmer need not worry about the interactions between the object they are constructing and other objects. This claim has not been subject to empirical evaluation but the analysis presented in this paper would suggest that such a claim may well be valid. One should note, however, that at other levels, the use of object-oriented languages may place an extra burden upon working memory. In particular, in most object-oriented systems one is forced to specify the relationships among objects (or more accurately, among classes of objects) before operations upon those objects can be defined. Détienne (1990) has shown that this requirement causes considerable difficulties since changing the structure of an evolving program can be very difficult. To avoid this problem, one might expect programmers to rely upon working memory in order to establish relationships between objects and classes before committing this structure to an external representation.

One issue that is important in the context of the display-based analysis proposed in this paper is how one might begin to devise a scheme for externalising working memory during program generation. In the realm of object-oriented languages it has been proposed that one way of facilitating this is to provide a description level similar to that found in bibliographic databases (Green, Gilmore, Blumenthal, Davies and Winder, in press). Here, the problem is one of retrieving a target from a partial description. Typically, bibliographic databases not only represent attributes of a text itself but also additional key words which can be used for searching and browsing. In programming terms, descriptors might be based upon persistent (eg.

functional subsystems) or transient (eg. a set of items to be documented) relationships between code structures; chronological relationships (eg. code developed or tested at a particular time) etc.

Providing this additional representation of a program may facilitate the recomprehension process that is central to the analysis presented in the present chapter. In particular, the provision of a description level could make certain relationships salient within a given task context, thus facilitating display-based recomprehension. Indeed, even such things as simple colour cues or tags (Lansdale, Simpson and Stroud, 1988) which identify commonalities between important code structures may facilitate the representation of salient relationships. Clearly there is significant scope for further research into mechanisms which can support both the externalisation of working memory and the recomprehension processes that appear to be central to display-based competence.

The language features described in this section are important in terms of the present analysis, since the incremental nature of code generation and comprehension/recomprehension will clearly be affected by the nature of the language. The present analysis extends existing work by suggesting ways in which language features and strategy may interact in concert with features of the task environment to give rise to particular forms of behaviour. It should be noted that these effects would not taken into account by display-based views, since the salience of particular features of the display remains undifferentiated. Hence, one important extension to the display-based models of problem solving would be the incorporation of a mechanism which allows one to specify the salience of particular display-based features.

The analysis presented here also suggests that problem solving success in programming will in part be determined by the nature of the task environment. Of particular importance will be the extent to which an environment supports nonlinear generation strategies and the ease with which changes to existing structures can be effected. These considerations may shed light upon the finding that the use of certain forms of programming environment can be frustrating for experienced programmers. For instance, Neal (1987 a and b) has conducted a number of studies exploring the efficacy of syntax-directed editors. Such editors provide syntactic

templates for particular structures. Hence, in Pascal, if the programmer inserts a 'begin' statement, a corresponding 'end' statement will be generated automatically. Neal found that experienced programmers frequently expressed dissatisfaction with such editors. Neal (1987a) comments that expert programmers "felt that to enter a program they had to do much more, both conceptually and physically because of the methods allowed for inserting and changing text ..." (p 100). Neal's findings together with those reported in this paper suggest that environments intended to support the coding process should provide the flexibility to support both incremental development and change.

11.7 Summary

The experiments reported in this chapter suggest that the development of expertise in programming is dependent upon the adoption of strategies for effectively utilising an external display. Moreover, these experiments have demonstrated that increased working memory capacity or availability is not a necessary prerequisite of skilled performance in this domain. Rather, skilled programmers appear to engage in closely linked cycles of code generation and evaluation activities. According to this model, code is generated in a fragmentary fashion and the display is used as a repository for recording intermediate solution steps. In addition, it has been argued that the success of this strategy will depend upon features of the programming language and upon the nature of the task environment.

While the analysis presented in this chapter has indicated the importance of display-based performance in programming, it has also suggested two primary limitations of this general approach. Firstly, existing accounts of display-based problem solving ignore the apparent relationship between expertise and the development of strategies for utilising display-based information. Secondly, such accounts do not consider the possibility that different forms of display-based information will be differentially salient in the context of a given task. Further developments of display-based accounts of problem solving will need to address these issues if they are to provide a full and coherent description of human problem solving in the context of complex tasks.

Chapter 12 Knowledge restructuring processes and the development of expertise in programming

12.1 Summary of experimental work

Two experiments were reported in chapter six which highlighted the effects of design experience upon the nature and development of programming plans. These experiments showed that the plan theory cannot account straightforwardly for the differences between novice and expert programming performance. In particular, they demonstrated the important role played by the acquisition of design-based knowledge in the comprehension of programs. The implications of these experiments are that current views concerning the nature and development of programming plans are flawed in two ways. On the one hand, the notational view is too narrow in its perspective because of the emphasis it places on notation at the expense of other demonstrably important factors. On the other hand, the views expounded by the Soloway group reflect a fundamental confusion between the measurement of plans and their use in theoretical explanations of expert performance. Hence, neither provides a sound theoretical basis for a full psychological theory of programming.

Subsequent experiments reporteded in this thesis suggested that the development of expertise in programming does not simply involve the accumulation of plans. Rather, programming expertise appears to depend upon the structuring of programming knowledge such that certain salient plan elements can be retrieved and accessed more quickly. It may be the case that design training facilitates this structuring process by encouraging programmers to focus upon the salient elements of plans. In addition, this might be expected to enhance the mapping between the language and problem domain that is discussed above, by providing a means of applying the salient features of plans and establishing the links between them.

The results of the experiment reported chapter seven extend this theme by providing general support for the idea that the possession of plans per se does not necessarily guarantee the same level of programming performance for different

groups of subjects. Hence, in the context of that experiment, while intermediates were to be able access the same plan knowledge as experts, they were able to achieve this more effectively, as evidenced by their greater speed.

The implications of this experiment are twofold. Firstly, it seems clear that a programmer's knowledge representation must support the representation of salient code structures. Such structures, which might be characterised as 'beacons' or 'focal lines', act as partial descriptions of particular code fragments and provide reminders that a segment of a program may need completing at a subsequent stage. In addition, the development of these code structures appears to coincide with increasing expertise. This may suggest that as expertise develops, knowledge structures change such that the organisation of these structures reflects the increasing importance of 'focal lines' and 'beacons' etc.

The results of the experiments presented in chapter 8 contribute to our understanding of the processes which underpin schema or plan development, and provide a framework for elaborating the relationship between the development of knowledge structures and expertise. One of the more interesting findings of the study presented there was that notation does not appear to support an opportunistic or a breadth-first strategy to the same degree for programmers of different skill levels.That is, the effects of notation on strategy are less extensive for experts than for intermediates.

One way to explain this differential effect might be to suggest that notation and knowledge representation interact very strongly to determine strategy. Hence, as representations of programming knowledge are in the process of development, as we suggest in the case of intermediates, then any additional means of facilitating programming strategy, such as might be provided by certain features of the notation, are likely to be of particular importance. At higher levels of skill, factors relating to the organisation of knowledge appear to play a greater part in the determination and the support of programming strategy.

A number of interesting findings emerged from this analysis. Firstly, it was shown that opportunistic episodes may occur at any point during the evolution of a program. However, this does not rule out the existence of an overall top-down

strategy. Hence, the clear dichotomy between top-down and opportunistic approaches that is implicated in previous work may be unfounded. Secondly, it was demonstrated that the emergence of top-down or opportunistic strategies is not task dependent as suggested by a number of previous studies. Rather such strategies can co-exist within the context of a single task. However, one form of strategy make take precedence over the the other at particular points during the evolution of the program.

Expertise also appears to play a major role in the determination of particular forms of strategy. For instance, novice programming behaviour appears to be systematically opportunistic, displaying none of the characteristics of a top-down approach. Conversely, expert programmers adopt a broadly top-down approach, at least during the early stages of program generation. In this context, the notion of the focal line appears to play a significant role in expert programming behaviour. The experiment reported in chapter 8 demonstrated that experts tend to generate focal lines first, providing a framework around which the rest of the program can be constructed. This may arise as a consequence of the knowledge restructuring process proposed in this thesis whereby the devlopment of expertise is seen to be accompanied by the restructuring of plan/schemata structures such that focal lines achieve promanance.

The results of the experiment reported in chapter 9 suggested that knowledge structures may develop via a restructuring process rather than through a process of knowledge accretion which simply involves developing a larger repertoire of plans. Previous accounts of programming expertise have tended to emphasise the development of extensive repertoires of programming knowledge rather than focus upon issues relating to knowledge restructuring processes. This is exemplified in Soloway's work on plans, where it is suggested that expertise simply involves building a more extensive collection of programming plans together with the rules which goven their use. However, as we have seen, the possession of plans per se cannot be used to diffrentiate certain groups of programmers, especially at higher levels of skill; namely at the point of transition between intermediates and experts.

The final experiment indicated the importance of display-based performance in programming and suggested two primary limitations of this general approach. Firstly, existing accounts of display-based problem solving ignore the apparent relationship between expertise and the development of strategies for utilising display-based information. Secondly, such accounts do not consider the possibility that different forms of display-based information will be differentially salient in the context of a given task. Further developments of display-based accounts of problem solving will need to address these issues if they are to provide a full and coherent description of human problem solving in the context of complex tasks. In the context of the present discussion it was suggested that focal lines would be diffirentially salient and would form a kernal around which code could be built. We suggested that the primary mechanism in code generation was based upon the close interaction between planning and re-evaluation which demands extensive reliance upon information that had previously been externalised.

12.2 An overview of the framework - Knowledge restructuring in programming

These experiments form the empirical basis for the central argument advanced in this thesis. This argument proposes that the possession of plans or of other schematic programming knowledge structures does not in itself provide an adequate account of the development of expertise in programming. The experiments reported in this thesis have demonstrated that programming expertise depends not only upon the possession of plans but also upon the structure of plan knowledge and upon the way in which such knowledge can be used to guide strategy. For example, the experiments reported here have shown that intermediate and expert programmers can display similar levels of performance in certain tasks which require the application of generic plan knowledge. However, in contrast, other features of performance appear to be dependent upon the level of expertise of the programmer. The results of these experiments have been taken to imply that while intermediates and experts may be able to access a similar range of plan structures, they appear to use these knowledge structures rather differently during both program generation and comprehension. The reasons for this appear to be related to the way in which this knowledge is structured.

To account for this phenomena, it has been argued that the development of programming expertise is underpinned by a knowledge restructuring process, leading to the development of hierarchically structured schemata which emphasise the salient aspects of each plan or schema structure. It has been suggested that this restructuring process may give rise to differences in the observed program generation strategies that are adopted by programmers of various skill levels. Hence, it has been demonstrated that expert programmers adopt a strategy whereby the focal aspects of plans are generated first and other program structures are built around them. This can be contrasted with other models of program generation, for example that presented by Rist, which predicts that experts will adopt a sequential mode of code generation and develop programs in schema order. The framework presented here has more in common with the parsing/gnisrap model advanced by Green et. al. which suggests that programs are developed in a fragmentary and incremental fashion. The parsing/gnisrap model also stresses the very close link between planning and execution which is a central element in the present analysis.

A number of the experiments reported in this thesis have demonstrated that expert programmers do not generate their programs in a linear fashion. More specifically, it appears that experts tend to generate the focal elements of plans and then later extend these to include subsidiary plan elements. It has been suggested that developing programs in this way reduces the demand on working memory since the decomposition of the emerging program will be held at a single level of abstraction (see chapters 8, 9 and 11). In chapter 11 it was suggested that jumping between different levels of abstraction in a problem space can place significant demands upon working memory (see also Anderson, 1983) and this may lead to the observation that experts tend to decompose their solutions in a broadly top-down fashion.

Experts also appear to rely extensively upon externalising the contents of working memory during program generation (see chapter 11). It was suggested in chapter 11 that one of the major determinants of expertise may be the adoption or the development of strategies which facilitate the effective use of external memory sources. As a consequence, it has been argued that both the nature of the

problem solving environment and the programming language will have an effect upon problem solving behaviour. In particular, given the incremental nature of program generation, the success of this process will in large part be determined by the flexibility of the environment in terms of the support it provides for implementing changes. Similarly, the viscosity of the language will affect this process, since it determines the ease with which changes can be made. In terms of comprehension, other factors come into play. Here, the expressiveness of the language, measured in terms of the ease with which various structures can be located and differentiated, will play a central role in facilitating comprehension.

The model of programming behaviour presented in this thesis suggests a small number of reasonably straightforward mechanisms which can account for both the generation and the comprehension of programs. In addition, this framework stresses the importance of both language and environmental features in determining the nature and success of programming behaviour.

In this final chapter, a more detailed framework for understanding the knowledge restructuring processes outlined in this thesis will be presented. This chapter will attempt to demonstrate how the experimental work reported in this thesis can be used to support a parsimonious interpretation of program generation and comprehension.

In the context of this framework there appear to be three important areas of concern. Firstly, we need to explain action. That is, how schematic representations of programming knowledge are instantiated as code. This derives from a consideration of the problems of previous schema-based accounts of programming behaviour and in particular the notion that such accounts have failed to provide a specification of the processes governing schema instantiation. This problem appears to derive largely from the fact that the components of schemata are equally accessible, and that they can be implemented in any order. It is argued that this leads to difficulties in understanding how action is initiated. Moreover, such models provide no indication of how programming strategy might develop. Indeed, they fail to provide any description of the nature of the strategic elements of problem solving in this complex domain. Hence, while such accounts may provide a theory of plans, they fail to provide a theory of planning.

The second area of concern discussed in this chapter relates to the actual representation of programming knowledge, and in particular to the form of representation one might adopt in order to provide a basis for a hierarchical representation of such knowledge. In this context, it is suggested that schemata are not flat structures where every part of the structure can be accessed equally. Adopting ideas from the text comprehension domain, it is suggested that schemata structures may be viewed as hierarchically structured propositional representations. In this context, the salient elements of programming plans, that is those elements that encode information relating to the current goal (focal lines) will occur at higher levels of the propositional hierarchy. It is suggested that this form of representation may go some way towards capturing the structural organisation of programming knowledge that is hypothesised to give rise to many of the experimental findings reported in this thesis.

Models of programming behaviour based upon propositional accounts are not new. However, such models have failed to provide a cogent explanation of how levels in a propositional hierarchy can be determined. Moreover, these models have not considered the way in which knowledge structures might develop with expertise. One aim of this thesis is to address these limitations by providing an account of the nature and development of programming knowledge and a demonstration of how differences in the structure of this knowledge can lead to differences in strategy.

Finally, consideration is given to the acquisition of structured representations of programming knowledge. It is suggested that the development of structured representations may not simply arise via the control of a basic psychological mechanism such as knowledge compilation. Rather, it is argued that the specific learning or training experience of the programmer contributes significantly to the development of hierarchical representations of programming knowledge.

12.3 Knowledge restructuring - A process model of schema instantiation

One important area of work which may contribute to the development of a model of knowledge representation in programming is that concerned with representations of linguistic knowledge and in particular with work in the text comprehension domain. It is clear that the plan theory of programming has strong parallels with work in these domains. In particular, advocates of the plan theory have attempted to frame their analysis of programming knowledge in terms of Schank's notion of scripts. Scripts have been used to provide a representational scheme for describing generic domain specific knowledge structures, and on this basis clearly bear some relationship to programming plans.

However, one of the main criticisms of Schank's theory is that is that it lacks a process model which specifies in detail how scripts or schemata are used. For example, Anderson (1983) claims that the symmetry of schemata can lead to potential problems "For instance, from the fact that the light is green one wants to infer that one can walk. One does not want to infer that the light is green from the fact that one is walking. No successful general-purpose programming language has yet been created that did not have an asymmetric conditionality built in as a basic property. We should expect no less of the human mind" (p 39). The problem with schemata is that it is very difficult to see how they can provide a basis for understanding action. Similarly, the plan theory of programming fails to specify how plans are used. It seems rather anomalous that researchers are prepared to advance a theory of plans without specifying a theory of planning. One of the main aims of this thesis has been concerned with explicating a framework to account for action and planning in programming, and this framework depends to a significant degree upon the rejection of some of the rather more simplistic notions which are embodied in plan theories.

The present framework suggests that expertise may involve the acquisition of declarative schemata which are accessible in the context of certain recognition and recall tasks (see chapter 10). Hence, programmers of different skill levels (expert and intermediate) can respond with similar levels of accuracy to the presentation of probe items, but experts tend to respond faster than intermediates in response to focal lines, whereas there is no difference in response to non-focal lines.

This finding suggests that while programmers can access their declarative programming knowledge, there is a basic asymmetry in their representation. The work presented here suggests a schema-based framework which may go some way toward addressing the criticisms typically levelled at other schema-based models of problem solving. In particular, as Anderson (1983) points out, in most schema based accounts, it is possible to instantiate and/or execute any part of a given schema since the control structure inherent in such models provides equality of access to all components. In the present context, it is suggested that certain elements of programming schemata will have permanently raised activation levels and will tend to be generated before other elements of the same schemata (see chapter 9).This reflects the empirical finding that focal lines tend to be generated before other elements of a particular schemata are expanded and/or refined.

12.4 Knowledge restructuring - towards a hierarchical model of knowledge representation in programming

In order to develop the model presented in this thesis in more detail it is necessary to not only specify the mechanisms which underpin the knowledge restructuring and control processes that are central to the theory, but also to outline a representational scheme which can accommodate a hierarchically organised knowledge base of plans and goal structures.

One way of viewing programming knowledge from this perspective is to consider such knowledge structures as being represented as a set of hierarchically structured propositions. This is the approach taken by Kintsch and van Dijk in their analysis of text comprehension and production.

Kintsch and van Dijk suggest that there are three levels of memory representation for text which can be distinguished. At one level, a text can be described in terms of the exact words and phrases used. Kintsch and van Dijk refer to this as the surface level representation. Another level of representation captures the semantic content of the text which represents both local (microstructure) and

global (macrostructure) features. The content of this level is captured in propositional form (Kintsch, 1974), where each statement in the text is represented as an individual proposition. For example, the phrase "If Mary Trusts John she is a fool" might be represented as (IF, (TRUST, MARY, JOHN),(FOOL, MARY)). Kintsch and van Dijk argue that this sentence would be represented by subjects in a hierarchical fashion with the proposition (IF, (TRUST, MARY, JOHN) occurring at a higher level than (FOOL, MARY). One prediction arising from this is that propositions identified as important (i.e., high in the propositional hierarchy) will be best recalled. A number of studies have shown this to be the case (Christiaansen, 1980; Kintsch and Keenan, 1973; Mandler and Johnson, 1977 and Meyer, 1975) and these studies provide strong evidence for some form of hierarchically structured representation of linguistic knowledge.

The experiment reported in chapter 10 employs a similar paradigm to analyse programming knowledge. Here, it was demonstrated that highly skilled programmers are able to respond more quickly to focal lines than intermediates, although their levels of accuracy were similar. Novices, however, performed much less accurately than both experts and intermediates. By analogy to the work of Kintsch and his colleagues, this study may suggest that certain salient programming structures may appear at higher levels in a programmer's macrostructure representation. Moreover, the restructuring of programming knowledge appears to be related explicitly to the development of expertise.

In terms of the framework presented here, plan structures in programming might be seen as a similar unit of analysis as the macrostructure representation of text as proposed by Kintsch. The macrostructure of a text represents sentences in terms of agents, goals and objects. It may be possible to derive a similar form of representation for programming knowledge where a focal line may be taken to represent or to express the goal of a particular plan. In terms of the present analysis, each goal structure will manipulate one more objects according a given procedure. In this sense a plan or programming schemata should be seen as analogous to a sentence. Previous analyses which have attempted to describe programming knowledge using the text structure model presented by Kintsch have described each program statement in propositional form. However, a more

appropriate level of representation would appear to suggest that programming plans or schemata should be considered as the basic unit of representation having more in common with the macrostructure level of analysis.

Having said this, however, the programming plan also appears to bear some resemblance to the third level of text structure knowledge proposed by Kintsch and van Dijk. Kintsch and van Dijk term this third level of representation the situation model. They claim that the situation model represents the situation described by the text and that it is "detached from the text structure proper and embedded in pre-established fields of knowledge" (p 135). They go on to suggest that "(T)he principle of organisation at this level may not be the text's macrostructure, but the knowledge schema (e.g., an appropriate script or frame) used to assimilate it." (p 135-136)

It is of interest to note that the situation model may appear to represent the same kind of structure as described by programming plans. However, this analogy poses problems if one examines the correspondence more closely. For example, it is clear that since notational factors appear to play a role in the perception and use of plans, then the text structure itself must in turn influence the representation of plan structures.

Atwood and Ramsey (1978) have also applied Kintsch and van Dijk's model of text comprehension to program understanding. They equate individual program statements with propositions, and the macrostructure representation is formed by grouping these propositions into a functional unit or chunk. Atwood and Ramsey, attempted to provide some evidence for their analysis by examining bug detection rates. They predicted that bugs residing at higher levels in the macrostructure hierarchy would be easer to detect than bugs at lower levels. Their data provides some support for this prediction and subsequent studies have found similar patterns of results (Vessey, 1989).

There are, however, a number of serious problems with this work. The main weakness of these studies is that the measure of depth in the proposition structure is flawed and does not correspond to Kintsch's propositional analysis. For example, Vessey uses depth in the control structure of the program as a measure

of level in the propositional hierarchy. However, Kintsch's propositional analysis attempts to capture the salience of each proposition in terms of its meaning to the reader. Another problem with this work is that, in Vessey's case at least, the conclusions that are drawn are based upon the analysis of a single bug occurring in different places. However, as Gilmore (in press) points out the 'same' bugs in different locations, may not be the same bugs at at. For example, Gilmore claims that while control-flow bugs may seem to be easily equated, the same cannot be said about semantic and plan-related bugs since they cannot be equivalent unless they appear at the same point in the program. A third problem, which may be related in part to these other difficulties, has been noted by Pennington (1985) who has suggested that the Atwood and Ramsey study may have been confounded by the location of bugs in the program text. It appears that bugs that resided high in their propositional hierarchy also occurred near the beginning of the program.

The model presented here attempts to specify the salience of programming structures at the program text level by equating focal lines with structures which occur at high levels in the propositional hierarchy. A focal line, as we have seen, describes the line of code that directly implements the programmer's current goal and hence we would expect that this structure would have some psychological significance for the programmer.

Gilmore (1988b) suggests that Schneiderman's (1980) model of programming is also very similar to Kintsch and van Dijk's model and that "the major weakness of this approach is that although it places much emphasis on factors of notational design (because the main process is the extraction of information from the program), it has nothing to say about what features of notations improve the efficiency of these processes, even though this should be one of the important functions of a model of program comprehension." (p 16). The work reported in this thesis represents a step in the opposite direction in that predictions about notational properties of programming languages are related to an explicit model of knowledge representation in programming. This model also makes predictions about the forms of strategy adopted by programmers of different skill levels and the extent to which certain notational properties might support preferred strategies.

In summary, while other models of programming have attempted to use ideas from the text comprehension domain none have paid sufficient attention to the saliency of particular programming structures nor to the effects of surface features of the text base that might affect program comprehension and generation. The main contribution of the present analysis is that it provides a means of specifying the saliency of particular structures and demonstrates how features of the notation of a language might affect the implementation and the subsequent comprehension of such structures.

12.5 Acquiring hierarchical representations: Design training and the development of structured representations of programming knowledge

One question that arises in terms of the above discussion is how the process of knowledge restructuring presented in this thesis might be initiated. It is clear that one of the central issues addressed in this thesis is the extent to which structured knowledge representations might develop via straightforward mechanisms of skill acquisition or alternatively whether they arise as a consequence of specific forms of training.

In chapter 6 it was demonstrated that certain kinds of tasks involving plan knowledge appear to depend upon the prior learning experiences of the programmers studied. In particular, it was shown that design experienced programmers performed significantly better in those situations which demanded the application of plan-based knowledge. It might be suggested on the basis of the studies reported in this chapter that design training facilitates plan use by focusing upon methods which emphasis the strict application of a top-down hierarchically levelled model.

In other domains it has been demonstrated that presenting hierarchically organised instructions can not only facilitate the acquisition of declarative knowledge (Smith and Goodman, 1984) but can also increase performance in complex problem solving tasks (Eylon and Reif, 1984; Larkin, 1980; Zeitz and Spoehr, 1989). Hence, it might be claimed that the degree to which explanatory

or training material emphasises a hierarchical structure will influence the development of knowledge representation which might in turn exert an influence over the degree to which that information can be proceduralised with practice into effective problem solving methods.

The effects of specific forms of prior training has not, until recently, been a factor that has figured significantly in more traditional accounts of skill acquisition. For example, in terms of the ACT* framework , programmers who achieve similar levels of practice should develop similar levels of procedural skill. Learning and skill acquisition in the ACT* framework are effectively based upon the amount of material presented and practised, while much less emphasis is placed upon the way in which this learning material is presented. The ACT* framework tends to emphasise the hierarchical nature of goal structures while neglecting specific issues in training. More recently, the embodyment of ACT* principles in intelligent tutoring systems has led to the suggestion that some effort should be put into communicating goal structures explicitly to students. Anderson, Boyle, Farrell and Reisner (1987) suggest that "there is a natural danger of casting instruction ... in terms of the linear structure (of the program)" (p 103). They go on to suggest that "fortunately, more enlightened instruction does emphasise a hierarchical, structured program" (p 103). However, they suggest "while structured programming is definitely a step in the right direction, it only ameriorates the basic problem ... the structured program itself is only a syntactic object which will have an imperfect correspondence to the structure of the programmer's plan" (p 103).

However, while seemingly excluding structured programming as a method of communicating goal structures, Anderson et al do not go on to suggest how a goal structure should be communicated. They qoute protocol studies which indicate that the explicit tutoring of plan structures can improve programming performance (Pirolli and Anderson, 1985), however they say nothing about how this might relete to instruction in the way in which plans are hierarchically linked and structured. The work reported in this thesis may go some way toward proving and account of the way in which specific forms of training might lead to improved problem solving performance.

In particular, it is argued that while structured programs may be linear artifacts which may not communicate a program's goal structure nor necessarily map onto the structure of programming plans, the actual process of devloping structured programs may encourage programmers to focus upon the salient features of plans and their application. Indeed, as we saw in chapter 6, the development of certain forms of programming skill (particularly the ability to use plan structures) appears to depend centrally upon the way in which learning material is presented to students. Hence, while the two groups of programmers studied in that experiment possessed equivalent levels of experience with the particular language concerned, they tended to use their knowledge rather differently. Moreover, another implication of this work is that programming plans are not taught directly to students. The efficacy of tutoring environments which emphasise the explicit teaching of plans, as suggested by Anderson et al, may be misguided in that it fails to recognise that an important element in skill development appears to be concerened with establishing the salient elements of plans and mapping between plans and actual code structures. It was suggested in chapter 6 that design training may facilitate these kinds of abilities.

The work reported in this thesis appears to draw into question important aspects of extant theories of skill acquisition to the extent that it suggests that certain forms of training may lead to hierarchically organised knowledge structures and consequently to differences in the way in which such knowledge is applied. It appears that in situations where learners are subject to the same amount of practice but where they are not given a framework for organising the knowledge that they acquire then their performance in certain tasks is significantly impaired. In the context of the present discussion, it might be argued that design training (particularly in structured programming/design) causes programmers to focus upon salient elements of relevant knowledge structures and that this in turn may lead to the development of structured schemata where focal plan elements are salient.

12. 6 Implications for further experimental work

The framework proposed in this thesis suggests a number of possibilities for further research into the cognitive processes that might be involved in programming. For example, one issue relates to debugging. The experiment reported by Vessey described above showed that bugs are not detected more quickly if they occur at higher levels in a program's propositional hierarchy. However, as we have seen, Vessey equates position in the structural hierarchy with the indentation level of the program statement. One prediction arising from the work reported here is that bugs in focal lines will be detected more quickly than bugs in occurring other structures. Moreover, focal lines may be highly embedded in the program's indentation structure, indicating a possible confounding factor in Vessey's study. If errors occurring in focal line are detected more quickly than errors residing elsewhere this would indicate that focal lines are of particular salience to programmers. This may also have more general implications for the design of tools intended to support the debugging process.

Figure 12.1 illustrates a propositional analysis of one of the experimental programs discussed in chapter 6. This program was analysed according to Kintsch's original formaulation (Kintsch, 1982). However, certain modifications were needed to accommodate the differences between programs and texts. Two major assumptions were made. Firstly, the notion of an *argument* in terms of textual analysis has been translated into the *value* of a variable. This may be slightly problematic since the value of this variable will change during execution. For instance, in figure 12.1 the value of the variables SUM and COUNT are used in the calculation of Average.

Under strict argument repetition, SUM and COUNT would be bound to the first occurrence of SUM and COUNT (as represented by the intialisation). In terms of the interpretation of argument repetition that is presented here, data flow is used to represent hierarchy as opposed to simply using the name of the variable. The choice of data flow was to some extent aribitrary, since control flow or an other mode of representation might have been used. The illustrates one difficulty of considering programs as analogous to text. In particular, there are many ways in

which to view a program, and each of these different perspectives is likely to give rise to a different representation of the problem.

Figure 12.1 presents this propositional analysis and indicates the program's focal lines (derived from Rist's (1989) definition). This diagram shows that, in terms of this analysis at least, focal lines are not represented with greater saliency than other structures in the propositional analysis. It may be that other forms of representation would show greater saliency for focal structures. Another line of research might be concerned with the ability of different forms of representation to provide support for the empirical findings presented in this thesis. For example, a more fine-grained analysis might show that, while focal-lines may not reside at a high level in the structural hierarchy, there may well still be systematic differences in recall when recall patterns are correlated with different levels in this structure. It is possible that we may not be able to construct macrostructure representations of programs since they may differ too significantly from text in their communicative function. It is clear that more empirical research will be required before the proposed analogy between text and programs can be supported.

One interesting observation that arose while constructing the propositional analysis presented below is that while focal lines may not be hierachically distinct, it is clear that, compared to other components of the program, they have many more links with other propositions. Hence, line *e* has four connections, while lines *f* and *d* have 3 connections. Note that with the exception of *h* and *j*, all other propositions have only two connections. It is possible that the degree of connectiveness is a better predictor of recall strategy, debugging performance and other behaviour than level in the propositional hierarchy.

Another area of potential interest stemming from the present analysis might be concerned with the question of whether there are performance changes at the point where knowledge restructuring takes place. For example, Lesgold et al (1988) have shown that many errors occur at the point where representations begin to change in the context of the development of a complex problem solving skill. This study demonstrated a nonmonotonic relationship between experience and performance in the development of radiological diagnostic skills. Lesgold et

al suggest that this kink in the learning curve might arise as a consequence of a major shift in processing or in knowledge representation. Immediately after a shift, intermediate practitioners were much more likely to make errors than they were before.

```
Program A
1     Sum = 0                          a INITIALISE (Sum 0)
2     Count = 0                        b INITIALISE (Count 0)
3     read (number)                    c  SUBCALL (read, number)
4  (F) WHILE number <> 9999 DO         d LOOP (number <> 0)
5        BEGIN
6  (F)     Sum = Sum + number          e UPDATE (Sum, Sum + number)
7  (F)     Count = Count + 1           f  INCREMENT (Count)
8           read (number)               g SUBCALL (read, number)
9        END
10    IF Count > 0                     h TEST (Count)
11      THEN                           i  COND (Count > 0)
12        BEGIN
13          average = Sum/Count        j  SET (average, Sum/Count)
14          writeln (average)          k SUBCALL (write, average)
15        END
16      ELSE                           l  COND (Count <= 0)
17          writeln ('no legal inputs) m SUBCALL (write, average)
```

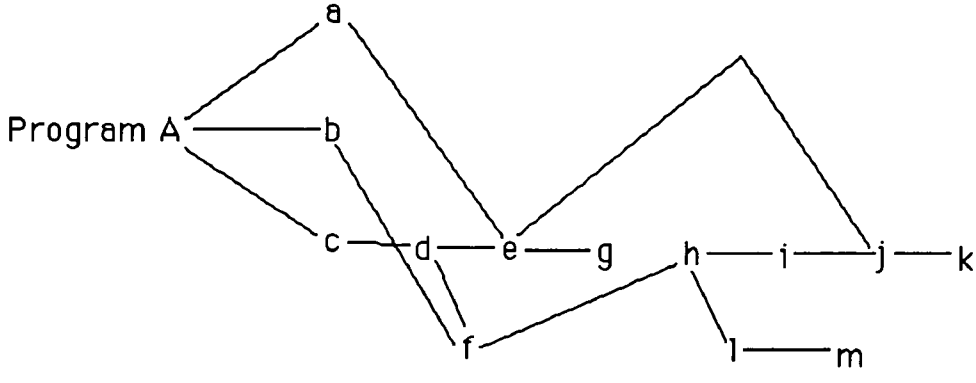


Figure 12.1 A propositional analysis of a short program. Focal lines are indicated in bold (**F**), and the propositional decomposition is shown on the right. The diagram illustrates the hierarchical level of each proposition (read from left to right). Note that the number of connections for focal lines is greater than for non-focal lines.

It should be noted that a similar phenomenon has also been reported by developmental psychologists (Bowerman, 1982; Karmiloff-Smith, 1979; Klahr, 1982; Strauss and Stavy, 1982). For instance Bowerman reported that children initially produce the correct instances of many irregular past-tense verbs and plural nouns (e.g., went, feet). Later they shift to incorrect regularisations of those words (e.g., goed, foots). Still later, children gain complete control over the irregularities in their vocabulary and stop making these kinds of mistakes. Hence, it seems that restructuring may lead to a short term decrement in performance in many areas of skill development when the transformation takes place between an adequate representation and an optimal representation. This might also be combined with procedural changes in the sense that strategies appropriate for one representation may be inappropriate in the context of another representation.

In terms of the present analysis, one might predict similar effects in the development of programming skill. An alternative possibility is that at intermediate skill levels processing is partly but not completely automated, with a fluctuating level of control suggesting that certain aspects of a task may be especially prone to error at this point. However, in the case of programming skill it may be the case that when such errors do occur they will be related more explicitly to focal structures in programs. If this is the case it would provide support for the idea that shifts in knowledge representation accompany the development of expertise and that the development of focal structures is an important part of this process.

Another possibility for further research is concerned with the question of whether there are other levels in the structural representation of programs. The present analysis suggests only two levels of representation, but it is possible that other important structures will be identified. The important point is that one should not consider schema structures as flat undifferentiated chunks of knowledge. Rather, it is clear that schema or plan structures will embody some internal organisation which reflects the importance of various structures to the programmer. The work reported here has identified the role of focal lines in capturing the salient goal structures of programming plans, but it is clear that other structures may also be of importance.

12.7 Implications for programming environments

In addition to these theoretical concerns, the model also poses a number of practical implications for the design of systems intended to provide support for the programming activity. For example, programming environments that incorporate a "fisheye" view of source code text (Furnas, 1986) rely upon some way of indicating a current focus around which the fisheye view can be constructed. Furnas (1986) suggests that a degree of interest (DOI) function can be assigned to each point in a program's structure indicating how interested a user is in seeing that point, given the current task.This DOI function is generated by an algorithm which employs a notion of 'a priori importance' to assign suitable values to points in the program structure. This a priori importance value is intended to represent components of a structure which are of psychological importance to a particular user engaged in a given task. However as Furnas concedes "the usefulness of a DOI ...will depend at least upon the suitable definition of ... a priori importance" (Furnas, 1986. pg 17). The findings of the present study may provide the basis for such a definition by indicating the salient features of a programmers knowledge representation. Hence, the identification of focal lines may provide the starting point for a more psychologically valid conception of a priori importance leading to the design of more suitable displays for complex information structures such as programs.

Of course for such a system to work one would need to automatically derive a program's plan structures and associated focal lines. However, this has not proved to be a straightforward task since commonly the same plan will not share the same surface characteristics at the code level (Rich and Wills, 1990). A promising approach has recently been proposed by Rist (forthcoming) who outlines a system intended to derive plan structure from code by tracing plan dependency links (capturing both data and control flow) from the plan focus to other elements of the plan. In its basic form, Rist's algorithm consists of a single recursive procedure that links the data flow into each line with its control flow. This algorithm employs a parsing mechanism - PARE (Plan Analysis by Reverse Engineering) which represents each basic system object (line of code) as a series of slots which when instantiated specify the use of that line of code, other code it

obeys, the data values it makes and the other lines of code that it controls (Rist and Bevemyr, 1991). Rist claims that this algorithm has been used on hundreds of Pascal programs to derive their constituent plan structure and as such it would seem like a potential contender for a system which might extract and display focal code elements and provide a basis for environments which can emulate or reflect cognitively-based structures.

12.8 Conclusions

In conclusion, this thesis has presented an analysis of the problem solving activities involved in programming tasks. A model has been presented which aims to extend our current thinking about problem solving in complex domains such as programming. This model emphasises the role of knowledge restructuring in the development of programming expertise and suggests that the processes of program generation and program comprehension are based upon the implementation of a small number of reasonably straightforward cognitive processes. These processes involve the creation or the location and subsequent recomprehension of the focal structures contained in schematic representations of programming knowledge.

The process of program generation is presented as an incremental process of fragmentary code generation and recomprehension which is governed by working memory limitations and is mediated by features of the programming language used and the environment in which the program is created. The process of program comprehension is described as an activity which involves the search for focal structures in extant code which can be mapped onto an internal representation of schematic programming knowledge. As for generation, the process of comprehension is also affected by language features in the sense that the ease with which focal structures can be discriminated will be dependent upon the the expressive power of the language. The processes of comprehension and generation should be seen as complementary since they are both fundamentally related to the development and the comprehension of salient structures residing either in the code or in terms of the programmer's representation of programming knowledge.

While this thesis has been concerned with problem solving activities in programming it clearly raises a number of issues with respect to the adequacy of existing generic models of problem solving. In particular, the work reported here suggests that a full understanding of problem solving in the context of complex tasks may require an articulation of the relationship between planning/operator selection and execution. The work reported in this thesis suggests that these processes cannot be considered in isolation. This in turn suggests that a significant amount of problem solving behaviour may only be explicable in terms of a detailed understanding of the way in which problem solvers externalise current states and then subsequently respond to these externalised states. It is suggested here that a more complete understanding of problem solving in complex domains will only be achieved once these factors begin to receive the attention that they deserve.

References

Adelson, B., (1981). Problem solving and the development of abstract categories in programming languages. *Memory and Cognition*, **9** (4), 422 - 433.

Adelson, B., (1984). When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory and Cognition*, **10**, 3, 483-495.

Adelson, B., (1985). Comparing natural and abstract categories: A case study from computer science. *Cognitive Science*, **9**, 417-430.

Adelson, B. and Soloway, E., (1985). The role of domain experience in software design. *IEEE Trans. SE* , **SE - 11** (11), 1351 - 1360.

Agre, P. and Chapman, D., (1987). Pengi: An implementation of a theory of activity. In Proceedings of the Sixth National Conference on Artificial Intelligence, 268 - 272. Menlo Park, Calif: American Association for Artificial Intelligence.

Allwood, C. M., (1984). Error detection processes in statistical problem solving. *Cognitive Science*, **8** 413 - 437.

Ambros-Ingerson, J. A., (1987). Relationships between planning and execution. *AISB quarterly newsletter*, **57**.

Anderson, J. R., (1976). Language, Memory and Thought. Lawrence Erlbaum, Hillsdale, NJ.

Anderson, J. R., (1982). Acqusition of cognitive skill. *Psychological Review*, **89**, 369 - 406.

Anderson, J. R., (1983). The architecture of cognition. Harvard University Press, Cambridge, MA.

Anderson, J. R., (1985). Cognitive Psychology and its Implications. W. H. Freeman, New York, NY.

Anderson, J. R., (1987). Acquisition of cognitive skill. *Psychological Review*, **89**, 369 - 406.

Anderson, J. R., (1989). The anological origins of errors in problem solving. In D. Klahr and K. Kotovsky (Eds.), Comple information processing: The impact of Herbert A. Simon. LEA, Hillsdale, NJ.

Anderson, J. R., Boyle, C. F., Farrell, R. and Reiser, B. J., (1987). Cognitive principles in the design of computer tutors. In P. Morris (Ed.), Modelling Cognition. John Wiley.

Anderson, J. R., Conrad, F. G. and Corbett, A. T., (1989). Skill Acquisition and the Lisp Tutor, *Cognitive Science*, **13**, 467 - 505.

Anderson, J. R., Greeno, J. G., Kline, P. J. and Neves, D. M., (1981) Acquisition of problem solving skill. In J. R. Anderson (Ed.), Cognitive skills and their acquisition. Lawrence Erlbaum, Hillsdale, NJ.

Anderson, J. R., Farrell, R. and Sauers, R., (1984). Learning to program in LISP. *Cognitive Science*, 8, 87 - 129.

Anderson, J. R. and Jeffries, R., (1985). Novice LISP errors: Undetected losses of information from working memory. *Human-Computer Interaction*, 1, 107-131.

Anderson, J. R., Kline, P. J. and Beasley, C. M., (1980). Complex learning processes. In R. E. Snow, P. A. Federico and W. E. Montague (Eds.), Aptitude, Learning and Instruction, Vol. 2. Lawrence Erlbaum, Hillsdale, NJ.

Anderson, J. R. and Reiser, B. J., (1985). The LISP tutor. *Byte*, 10, 159 - 175.

Anzai, Y. and Simon, H., (1979). The theory of learning by doing. *Psychological Review*, 86, 124-180.

Arblaster, A., Sime, M. E. and Green, T. R. G., (1979). Jumping to some purpose. *The Computer Journal*, 22, 105 - 109.

Atwood, M. E., Masson, M. E. and Polson, P., (1980). Further exploration with a process model for water jug problems. *Memory and Cognition*, 8, 182 - 192.

Atwood, M. E. and Polson, P. G., (1976). A process model for water jug problems. *Cognitive Psychology*, 8, 191 - 216.

Atwood, M. E. and Ramsey, H. R., (1978). Cognitive structure in the comprehension and memory of computer programs: An investigation of computer program debugging. Technical Report, ARI TR-78-A210, Science Applications, Englewood, CA.

Badre, A. N. and Allen, J., (1989). Graphic language representation and programming behavior. In G. Salvendy and M. J. Smith (Eds.), Designing and using Human-Computer Interfaces and Knowledge based systems. Elsevier, Amsterdam.

Barfield, W., (1986). Expert-novice differences for software: implications for problem-solving and knowledge acquisition. *Behaviour and Information Technology*, 5 (1), 15 - 29.

Bateson, A. G., Alexander, R. A. and Murphy, M. D., (1987). Cognitive processing differences between novice and expert computer programmers. *Int. J. of Man-Machine Studies*, 26, 649-660.

Bellamy, R. K. E. and Gilmore, D. J., (1990). Programming Plans:Internal or External Strucures. In K. J. Gilhooly, M. T. G. Keane, R. H. Logie and G. Erdos, eds., Lines of Thinking: Reflections on the Psychology of Thought ,Vol 1, Wiley.

Biermann, A. W., Ballard, B. W. and Sigmon, A. H., (1983). An experimental study of natural language programming. *Int. J. Man-Machine Studies,* 18, 71 - 87.

Black, J. B., Kay, D. S. and Soloway, E., (1987). Goal and plan representations: from stories to text editors and programs. In Interfacing thought: cognitive aspects of HCI, J. M. Carroll (Ed.), MIT Press.

Bonar, J., (1985). Understanding the bugs of novice programmers. Dissertation, University of Massachusetts, Amherst, MA.

Bonar, J. and Cunningham, R., (1988). Bridge: an intelligent tutor for thinking about programming. In J. Self (Ed.), Artificial Intelligence and Human Learning. Chapman Hall, London.

Bonar, J. and Liffick, B. W., (1990). A visual programming system for novices. In S. K. Chang (Ed.), Principles of visual programming systems. Pentice-Hall.

Bower, G. H., Black, J. B. and Turner, T., (1979). Scripts in memory for texts. *Cognitive Psychology*, **11**, 177 - 220.

Bowerman, M., (1982). Starting to talk worse: Clues to language acquisition from childre's late speech errors. In S. Strauss (Ed.), U-Shaped behavioural growth, Academic Press, New York.

Boyle, T. and Drazkowski, B., (1989). Exploiting natural intelligence: Towards the development of effective environments for learning to program. In A. Sutcliffe and L. Macaulay (Eds.), People and Computers V. Cambridge University Press, Cambridge.

Bransford. J. D. and Franks, J. J., (1971). The abstraction of linguistic ideas. *Cognitive Psychology*, **2**, 331 - 350.

Brayshaw, M. and Eisenstadt, M., (1988). Adding data and procedure abstraction to the Transparent Prolog Machine (TPM). Open University HCRL Research Rep. Number 31, Jan. 1988.

Brayshaw, M. and Eisenstadt, M., (1989). A practical tracer for Prolog. Open University HCRL Research Report Number 42. Feb. 1989.

Brooks, R. E., (1977). Towards a theory of the cognitive processes in computer programming. *Int. J. of Man-Machine Studies*, **9**, 737-751.

Brooks, R. E., (1983). Towards a theory of the comprehension of computer programs. *Int. J. Man - Machine Studies*, **18**, 543 - 554.

Bouma, H. and deVoogd, A. H., (1974). On the control of eye saccades in reading. *Vision Research*, **14**, 273 - 284.

Burton, R. R., (1982). Diagnosing bugs in simple procedural skill. In D. H. Sleman and J. S. Brown (Eds.), Intelligent Tutoring Systems. Academic Press, New York.

Brown, J. S. and Burton, R. R., (1978). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, **2**, 155 - 192.

Brown, J. S. and VanLehn, K., (1980). Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, **4**, 379 - 426.

Card, S. K., Moran, T. P. and Newell, A., (1983). The psychology of human-computer interaction. Lawrence Erlbaum, Hillsdale, NJ.

Carey, L., Flower, L., Hayes, J. R., Schriver, K. and Haas, C., (1987). Differences in writers' initial task representations (office of navel research technical report 2), Pittsburgh, PA: Carnegie Mellon University.

Carlson, R. A., Khoo, B. H., Yaure, R. G. and Schneider, W., (1990). Acquisition of problem solving skill: Levels of organisation and use of working memory. *Journal of Experimental Psychology: General*, 119, 2, 193 - 214.

Carpenter, P. A. and Just, M. A., (1989). The role of working memory in language comprehension. In D. Klahr and K. Kotovsky (Eds.), Comple information processing: The impact of Herbert A. Simon. LEA, Hillsdale, NJ.
Chapman, D., (1987). Planning for conjunctive goal. *Artificial Intellegence*, **32**, 333- 377.

Chapman, D., (1989). Penguins can make cake. AI Magazine, AAAI, Winter 1989. 45 - 50.

Chase, W. G. and Ericsson, K. A., (1982), Skill and working memory. In G. H. Bower (ed.), The psychology of learning and motivation. Academic Press, New York.

Chase, W. G. and Simon, H. A., (1973). Perception in chess. *Cognitive Psychology*, **4**, 55 - 81.

Cheng, P. W., (1985). Restructuring versus automaticity: Alternative accounts of skill acqusistion. *Psychological Review*, **92**, 3, 414 - 423.

Chi, M. T. H., Feltovich, P. J. and Glaser, R., (1981). Categorisation and preresentation of physics problems by experts and novices. *Cognitive Science*, **5**, 121 -152.

Chi, M. T. H., Glaser, R. and Rees, E. (1981). Expertise in problem solving. In advances in the psychology of human intelligence, Vol. 1. Lawrence Erlbaum, Hillsdale, NJ.

Christiaasen, R. E., (1980). Prose memory: Forgeting rates for memory codes. *Journal of Experimental Psychology. Human Learning and Memory*, **6**, 5, 611 - 619.

Cirilo, R. K. and Foss, D. J., (1980). Text structure and reading time for sentences. *Journal of Verbal Learning and Verbal Behavior*, **19**, 96 - 109.

Collins, A., (1978). Explicating the tacit knowledge in teaching and learning. Technical Report 3889, Bolt, Beranek and Newman, Cambridge, MA.

Collins, A. M. and Loftus, E. F., (1975). A spreading-activation theory of semantic processing. *Psychological Review*, **82**, 407 - 428.

Cohen, B. H., (1966). Some-or-none characteristics of coding behavior. *Journal of Verbal Learning and Verbal Behavior*, **5**, 182 - 187.

Cooke, N. M., Durso, F. T. and Schvaneveldt, R. W., (1986), Recall and measures of memory organization. Journal of Experimental Psychology: Learning, Memory and Cognition, 12, 538 - 549.

Cooke, N. J. and Schvaneveldt, W., (1988). Effects of computer programming experience on network representations of abstract programming concepts. *Int. J. Man - Machine Studies*, **29**, 407-427.

Crosby, M. and Stelovsky, J., (1989). Subject differences in the reading of computer algorithms. In G. Salvendy and M. J. Smith (Eds.), Designing and using Human-Computer Interfaces and Knowledge based systems. Elsevier, Amsterdam.

Crowder, R. G., (1976). Principles of learning and memory. Lawrence Erlbaum, Hillsdale, NJ.

Cunniff, N. and Taylor, R. P., (1987). Graphics and learning: A study of learner characteristics and comprehension of programming languages. Proc. of INTERACT'87, Stuttgart, 1987.

Cunniff, N. and Taylor, R. P., (1987). Graphical vs. textual representation: an empirical study of novices' program comprehension. In G. M. Olson, S. Sheppard and E. Soloway (Eds.), Empirical studies of programmers: second workshop, Ablex, Norwood, NJ.

Cunniff, N., Taylor, R. P. and Black, J. B., (1986). Does programming language affect the type of conceptual bugs in beginners' programs? A comparison of FPL and Pascal. In Human Factors in Computing Systems, Proc CHI'86, North-Holland, Amsterdam.

Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A. and Love, T., (1979). Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *IEEE Transactions on software engineering*, **SE -5** (2), 96 - 104.

Dahl, O. -J., Dijkstra, E. W. and Hoare, C. A. R., (1972). Structured programming. Academic Press, London.

deGroot, A. D., (1965). Thought and choice in chess. Moulton, The Hague.

Dearholt, D. W., Schvaneveldt, R. W. and Durso, F. T., (1985). Properties of networks derived from proximities. Memorandum in Computer and Cognitive Science, MCCS-85-14. Computing Research Laboratory, New Mexico State University.

Détienne, F., (1985). Programming expertise and program understanding. Paper presented at the 9th. Congress of the international ergonomics association. Bounemouth, Sept. 1985.

Détienne, F., (1986). Program understanding and knowledge organization: the influence of acquired schemata. Paper presented at the 3rd. European conference on cognitive ergonomics. Paris, Sept. 1986.

Détienne, F., (1989). A schema-based model of program understanding. To appear in, Mental models and Human-Computer Interaction, North-Holland. (presented at the 8th. international workshop on informatics and psychology, Scharding, Austria, May 1989).

Détienne, F., (1990). Difficulties in designing with an object-oriented language: an empirical study. In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.), Human-Computer Interaction, Interact'90. North-Holland, Amseterdam.

Détienne, F. and Soloway, E., (1989). Program understanding as an expectation-druven activity. In G. Salvendy and M. J. Smith (Eds.), Designing and using Human-Computer Interfaces and Knowledge based systems. Elsevier, Amsterdam.

Détienne, F. and Soloway, E., (1990). An empirically-derived control structure for the process of program understanding. *International Journal of Man-Machine Studies*, **33**, 323-342.

Dijkstra, E. W., (1968). Letter to the editor. *Communications of the Association for Compuing Machinery*, **11**, 147 - 148.

Draper, S. W., (1986). Display managers as the basis for user machine communication. In D. A. Norman and S. W. Draper (Eds.), User Centred System Design. Lawrence Erlbaum, Hillsdale, NJ.

Du Boulay, B. and Matthew, I., (1984). Fatal error in pass zero: how not to confuse novices. *Behaviour and Information Technology*, 3 (2), 109 -118.

Dyck, J. and Auernheimer, B., (1989). Comprehension of Pascal statements by novice and expert programmers, Poster Presented at CHI'89, Seattle, Washington.

Dyck, J. L. and Mayer, R. E., (1985). BASIC Versus Natural Language: Is There One Underlying Comprehension Process? Human Factors in Computing Systems. Proc. CHI'85. L. Borman and B. Curtis (Eds.), North-Holland, Amsterdam.

Egan, D. E. and Schwartz, B. J., (1979). Chunking in recall of symbolic drawings. *Memory and Cognition*, 7, 149 - 158.

Ehrlich, K. and Soloway, E., (1984). An empirical investigation of the tacit plan knowledge in programming. In J. C. Thomas and M. L. Schneider, (Eds.), Human factors in computing systems, Ablex, Norwood, New Jersey.

Eisenstadt, M., Breuker, J. and Evertsz, R., (1984). A cognitve account of 'natural' looping strategies. In the Proceedings of INTERACT'84, Elsevier, Amsterdam.

Elio, R., (1986). Representation of similar well-learned cognitive procedures. *Cognitive Science*, 10, 41 - 73.

Ernst, G. W. and Newell., (1969). GPS: A case study in generality and problem solving. Academic Press, New York.

Eylon, B. and Reif, F., (1984). Effects of knowledge organisation on task performance. *Cognition and Instruction*, 1, 5 - 44.

Fikes, R. E. and Nilsson, N. J., (1971). STRIPS: A new approach to the application of problem solving. Artificial Intelligence, 2, 189 - 208.

Fitter, M. J. and Green T. R. G., (1979). When do diagrams make good computer languages? *Int. J. Man-Machine Studies*, 11, 235 - 261.

Flower, L. & Hayes, J. R. (1980). The dynamics of composing: Making plans and juggling constraints. In G. Steinberg (Ed.), Cognitive processes in writing. Lawrence Erlbaum, Hillsdale, NJ.

Furnis, G. W., (1986). Generalized Fisheye Views. In M. Mantei and P. Orbeton (Eds.), Proc. CHI'86. Boston, MA. North-Holland.

Galotti, K. M. and Ganong, W. F., (1988). What non-programmers know about programming: Natural language procedure specification. *Int. J. Man-Machine Studies*, 22, 1 - 10.

Gentner, D. and Gentner, D. R., (1983). Flowing waters or teeming crowds: mental models of electricity. In D. Gentner and A. L. Stevens (Eds.), Mental Models. Lawrence Erlbaum, Hillsdale, NJ

Gilmore, D. J., (1986). Structural visibility and program comprehension. In People and computers: designing for usability. M. D. Harrison and A. F. Monk (Eds.), Cambridge University Press, Cambridge.

Gilmore, D. J., (1986b). The perceptual cueing of the structure of computer programs. Unpublished Ph.D. thesis, University of Sheffield.

Gilmore, D. J., (1990). Methodological issues in the study of programming. In J.-M. Hoc, T.R.G. Green, R. Samurcay and D. J. Gilmore, (Eds.), Academic Press, London.

Gilmore, D. J., (1990). Expert programming knowledge: A strategic approach. In J.-M. Hoc, T.R.G. Green, R. Samurcay and D. J. Gilmore, (Eds.), Academic Press, London.

Gilmore, D. J., (in press). Models of debugging. ACTA Psychologica.

Gilmore, D. J. and Green, T. R. G., (1984). Comprehension and recall of miniature programs. *Int. J. Man - Machine Studies*, **21**, 31 - 48.

Gilmore, D. J. and Green, T. R. G., (1987). Are 'programming plans' psychologically real - outside Pascal? Proc. of INTERACT'87, H. J. Bullinger and B. Shackel (Eds.), Elsevier Science Publishers B. V., North-Holland.

Gilmore, D. J. and Green, T. R. G., (1988). Programming Plans and Programming Expertise. *QJEP*, **40A** (3), 423 - 442.

Gitomer, D. H., (1984). A cognitive analysis of a complex troubleshooting task. Unpublished doctoral dissertation, University of Pittburgh, Pittsburgh, PA, USA.

Goldberg, A. and Robson, D., (1983) Smalltalk-80: The language and its implementation, Addison-Wesley, Reading, MA.

Gray, W. D. and Anderson, J. R., (1987). Change-episodes in coding: when and how do programmers change their code? In G. M. Olson, S. Sheppard and E. Soloway (Eds.), Empirical studies of programmers: second workshop, Ablex, Norwood, NJ.

Green, T. R. G., (1977). Conditional program statements and their comprehensability by professional programmers. *J of Occup. Psychol.*, **50**, 93 - 109.

Green, T. R. G., (1980). Programming as a cognitive activity. In Human Interaction with Computers, H. T. Smith and T. R. G. Green (Eds.), Academic Press, London.

Green, T. R. G., (1982). Pictures of programs and other processes, or how to do things with lines. *Behaviour and Information Technology*, **1** (1), 3 - 36.

Green, T. R. G., (1989). Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay (Eds.), People and Computers V. Cambridge University Press, Cambridge

Green, T. R. G., (1990a). Programming Languages as Information Structures. In J.-M. Hoc, T. R. G. Green, R. Samurcay and D.J. Gilmore (Eds.) The Psychology of Programming, Academic Press, London.

Green, T. R. G., (1990b). Representing and Manipulating programming knowledg. Working paper for CogBrow meeting. October 1990.

Green, T. R. G., (1990c). The cognitive dimension of viscosity: a sticky problem for HCI. In D. Diaper, D. Gilmore, G, Cockton and B, Shackel (Eds.) Human Computer Interaction - INTERACT'90., North-Holland, Amsterdam.

Green, T. R. G., (1991). Describing informtion artifacts with cognitive dimensions and structure maps. Paper to be presented at HCI'91

Green, T. R. G., Bellamy, R. K. E. and Parker, J. M., (1987). Parsing and gnisrap: a model of device use, Proc. INTERACT'87, H. J. Bullinger and B. Shackel (Eds.), Elsevier Science Publishers B. V., North-Holland.

Green, T. R. G. and Borning, A., (1990). The Generalised unification Parser: Modelling the parsing of notations. In D. Diaper, D. Gilmore, G, Cockton and B, Shackel (Eds.) Human Computer Interaction - INTERACT'90., North-Holland, Amsterdam.

Green, T. R. G. and Cornah, A. J., (1984). The programmer's torch. Proc. INTERACT'84. London 1984.

Green, T. R. G., Gilmore, D. J. and Winder, R., (1990). Towards a cognitive browser for OOPS: the program as a rich information structure. Paper presented at a GMS-IPSI workshop on Intelligent access to information systems. Darmstadt, Germany, November 1990.

Green, T. R. G., Gilmore, D. J., Blumenthal, B. B., Davies, S. P. and Winder, R., (1992). Towards a cognitive browser for OOPS. International Journal of Human-Comuter Interaction, **4(1)**, 1-34.

Green, T. R. G., Sime, M. E. and Fitter, M. J., (1980). The problems the programmer faces, *Ergonomics*, **23** (9), 893 - 907.

Green, T. R. G. and Payne, S. J., (1982). The Woolly Jumper: Typographic Problems of Concurrency in Information Display. *Visible Language*, **16** (4), 391 - 403.

Green, T. R. G. and Payne, S. J., (1984). Organization and learnability in computer languages. *Int. J. Man - Machine Studies*, **21**, 7 - 18.

Green, T. R. G., Petre, M. and Bellamy, R. K. E., (1991). Comprehensibility of visual and textual programs: a test of Superlativism agains the'match-mismatch conjecture. To appear in proceedings of ESP'91, New Brunswick, NJ.

Greeno, J. G., (1978). Nature of problem solving abilities. In W. K. Estes (Ed.), Handbook of learning and cognitive processes, Lawrence Erlbaum, Hillsdale, NJ.

Greeno, J. G. and Simon, H. A., (1974). Processes for sequence production. *Psychological Review, 81*, 187 - 181.

Guindon, R., (1988). Software design tasks as ill-structured problems, software design as an opportunistic process. MCC Technical Report STP-214-88.

Guindon, R., (1989). The process of knowledge discovery in systems design. In G. Salvendy and M. J. Smith (Eds.), Designing and using Human-Computer Interfaces and Knowledge based systems. Elsevier, Amsterdam.

Guindon, R., (1990). Knowledge exploited by experts during software system design. *Int. J. Man - Machine Studies*, **33**, 279 - 304.

Guindon, R., Krasner, H. and Curtis, B., (1987 a). Cognitive processes in software design: Activities in early, upstream design. Proc. INTERACT'87, Stuttgart, 1987.

Guindon, R., Krasner, H. and Curtis, B., (1987 b).Breakdowns and processes during the early activities of software design by professionals. In G. M. Olson, S. Sheppard and E. Soloway (Eds.), Empirical studies of programmers: second workshop, Ablex, Norwood, NJ.

Haas, C., (1987). How the writing medium shapes the writing process: Studies of writers composing with pen and paper and word processing. Doctoral dissertation in rehetoric. Pittsburgh, PA: Carnegie-Mellon University.

Haas, C., (1989). Does the medium make a difference? Two studies of writing with pen and paper and with computers. Human Computer-Interaction, 4, 2, 149 - 169.

Hass, C. and Hayes, J. R., (1986). What did I just say? Reading problems in writing with a machine. Research in the teaching of English, 20 (2), 22 - 35.

Haberlandt, K. (1980). Story grammers and the reading time of story constituents. *Poetics*, **9**, 99-116.

Haberlandt, K., Berian, C. and Sandson, J., (1980). The episode schema in stroy processing. *Journal of Verbal Learning and Verbal Behavior,* **19**, 635 - 650.

Halstead, M, M., (1977). Elements of software science. Elsevier, New York, NY.

Hayes, J. R., (1989), Writing research: The analysis of a very complex task. In D. Klahr and K. Kotovsky, (Eds.), Complex Information Processing; The impact of Herbert A. Simon. Erlbuam, Hillsdale, NJ.

Hayes-Roth, B. and Hayes-Roth, F., (1979). A cognitive model of planning, *Cognitive Science*, **3**, 275 - 310.

Hayes-Roth, B., Hayes-Roth, F., Rosenschein,S. and Cammarata, S., (1979). Modeling planning as an incremental, opportunistic process process. Proceedings of the Sixth International Joint Conference on Artificial Intelligence, Tokyo, Japan.

Hitch, G. J., (1978). The role of short-term working memory in mental arithmetic, *Cognitive Psychology*, **10**, 302 - 323.

Hitch, G. J. and Baddeley, A. D., (1976). Verbal reasoning and working memory, *Quarterly Journal of Experimental Psychology*, **28**, 603 - 621.

Hoc, J-M. (1981). Planning and direction of problem solving in structured programming: an empirical comparision between two methods. *International Journal of Man-Machine Studies*, **15**, 363 - 383.

Hoc, J-M., (1988). Towards effective computer aids to planning in computer programming: Theoretical concern and empirical evidence drawn from assessment of a prototype. In G. C. Van der Veer, T. R. G. Green, J-M, Hoc & D. M. Murray (Eds.), Working with computers: Theory versus outcome. Academic Press, London.

Hoc, J-M., Guyard, J., Quere, M. and Jacquot, J. P., (1984). Designing and evaluating computer aids in structured programming. Proc. INTERACT'84. London 1984.

Hoc, J. -M., (1988). Cognitive psychology of planning. Academic Press, London.

Howes, A. and Payne, S. J., (1990). Display-based competence: towards user models for menu-driven interfaces. *International Journal of Man-Machine Studies*, 33, 637 - 655.

Hunt, E. B. and Poltrack, S. E., (1974). The mechanisms of thought. In B. H. Kantowitz (Ed.), Human Information Processing: Tutorials in performance and Cognition, Lawrence Erlbaum, Hillsdale, NJ.

Intons-Peterson, M. and Fournier, J., (1986). External and internal memory aids: When and how often do we use them?. *J. Exp. Psychol. General,* 115, 267 - 280.

Jeffries, R. A. (1982). Comparison of Debugging Behavior of Novice and Expert Programmers. Pittsburgh, PA: Department of Psychology, Carnegie-Mellon University.

Jeffries, R., Polson, P., Razran, L. and Atwood, M. E., (1977). A process model for missionaries-cannibals and other river-crossing problems. *Cognitive Psychology*, 9, 412 - 440.

Jeffries, R., Turner, A. A., Polson, P. G. and Atwood, M. E., (1981). The processes involved in designing software. In cognitive skills and their acquisition, J. R. Anderson (Ed.), Lawrence Erlbuam, Hillsdale, NJ.

Johnson, W. L., (1988). Modelling programmers' intentions. In J. Self (Ed.), Artificial Intelligence and Human Learning. Chapman and Hall, London.

Johnson, W. L., (1990). Understanding and Debugging Novice Programs. *Artificial Intelligence*, 42, 51 - 97.

Johnson, W. L. and Soloway, E., (1985). PROUST: Knowledge based program understanding. *IEEE Trans. on SE.* SE - 11(3), 267 - 275.

Johnson, W. L., Soloway, E., Cutler, B. and Draper, S., (1983). Bug catalog 1. Research Report No 286, Computer Science Dept., Yale University, Yale, New Haven, Connecticut.

Just, M. A. and Carpenter, P. A., (1980). A theory of reading: From eye fixation to comprehension. *Psychological Review*, 87, 329 - 354.

Kaebling, L. P., (1987). An architecture for intelligent reactive systems. In M. Georgeff and A. L. LAnsky (Eds.), Reasoning about actions and plans, Morgan Kaufmann, Palo Alto, CA.

Kahney, H., (1983). What do novice programmers know about recursion. In A. Janda (Ed.), Proc. CHI'83. Boston, MA. North-Holland.

Kant, E. and Newell, A., (1984). Problem Solving Techniques for the Design of Algorithms. *nformation Processing and Management*, 20, 1-2, 97 - 118.

Kaplan, C. A. and Simon, H. A., (1990). In search of insight. *Cognitive Psychology*, 22, 374 - 419.

Karmiloff-Smith, A., (1979). Micro- and macrodevelopmental changes in language acquisition and other representational systems. *Cognitive Science*, 3, 91 - 118.

Katz, J. R. and Anderson, J. R., (1988). Debugging: an analysis of bug-location strategies. *Human - Computer Interaction*, 3, 351 - 399.

Kay, M. (1985). Parsing in functional unification grammer. In D. R. Dowty, L. Karttunen & A. M. Zwicky (Eds.), Natural Language Parsing: Psychological, computational and theoretical perspectives. Cambridge University Press.

Kay, D. S. and Black, J. B., (1984). The changes in knowledge representation of computer systems with expertise. Proceedings of the Human Factors Society, 28th, Annual Meeting, Santa Monica, USA.

Keane, M., Kahney, H. and Brayshaw, M., (1989). Simulating analogical mapping difficulties in recursion problems. Open University HCRL Research Report Number 41. Feb. 1989.

Kempen, G. and Vosse, T., (1989). Incremental syntactic tree formation in human sentence processing: an interactive architecture based on activation decay and simulated annealing. Cahiers de la Fondation Archives Jean Piaget.

Kemper, S., (1982). Filling in the missing link. *Journal of Verbal Learning and Verbal Behavior*, 21, 99 - 107.

Kernighan, B. and Plauger, P., (1987). The elements of programming style. McGraw-Hill, New York, NY.

Kesler, T. E., Uram, R. B., Magareh-Abed, F., Fritzsche, A., Amport, C. and Dunsmore, H. E., (1984). The effect of indentation on program comprehension. *Int. J. Man - Machine Studies*, 21, 415 - 428.

Kessler, C. M and Anderson, J. R., (1986). A model of novice debugging in LISP. Empirical Studies of programmers, First Workshop. Ablex, Norwwod, NJ.

Kieras, D. E. and Bovair, S., (1981). Strategies for abstracting main ideas from simple technical prose. Technical report no. UARZ/DP/TR-81/9, University of Arizona.

Kintsch, W., (1974). The representation of meaning in memory. Lawrence Erlbaum, Hillsale, NJ.

Kintsch, W., (1982). Memory and Cognition, Kriger, Malabar, Florida.

Kintsch, W. and Keenan, J. M., (1973). Reading rate and retention as a function of the number of propositions in the base structure of sentences. *Cognitive Psychology*, 5, 257 - 274.

Kintsch, W., Kozminsky, E., Streby, W. L., McKoon, G. and Keenan, J. M., (1975). Comprehension and recall of text as a function of content variables. *Journal of Verbal Learning and Verbal Behavior*, 14, 196 - 214.

Kintsch, W. and van Dijk, T. A., (1978). Toward a model of text comprehension and production. *Psychological Review*, 55, 363 - 394.

Klahr, D., (1982). Nonmonotone assessment of monotone development: An information processing analysis. In S. Strauss (Ed.), U-Shaped behavioural growth, Academic Press, New York.

Kolodner, J. L., (1983). Towards an understanding of the role of experience in the evolution from novice to expert. *International Journal of Man-Machine Studies*, 19, 497 - 518.

Laird, J. E., Newell, A. and Rosenbloom, P. S., (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33, 1 - 64.

Lange, B. M. and Moher, T. G., (1989). Some strategies for reuse in an object-oriented programming environment. In K. Bice and C. Lewis (Eds.), Proceedings of CHI'89. Addison-Wesley.

Lansdale, M. W., Simpson, M. and Stroud, T. R. M., (1988). A comparison of words and icons as external memory aids in an information retrieval task. *Behaviour and Information Technology*, 9, 2, 111 - 131.

Larkin, J. H., (1980). Skilled problem solving in physics: a hierarchical planning model, *Journal of Strucural Learning*, 6, 271 - 297.

Larkin, J. H., (1981). Enriching formal knowledge: A model of learning to solve problems in physics. In J. R. Anderson (ed). Cognitive Skills and Their Acquisition. Lawrence Erlbaum Associates, Hillsdale, NJ.

Larkin, J., (1989). Display-based problem solving. In D. Klahr and K. Kotovski (Eds.), Complex information processing: A tribute to Herbert A. Simon. Lawrence Erlbaum, Hillsdale, NJ.

Larkin, J. H., McDermott, J., Simon, D. and Simon, H. A., (1980). Expert and novice performance in solving physics problems. *Science*, 208, 1335 - 1342.

Lesgold, A. M., (1984). Human skill in a computerized society: complex skills and their acquisition, *Behaviour Research Methods, Instruments and Computers*, 16 (2), 79 - 87.

Lesgold, A., Rubinson, H., Feltovich, P., Glaser, R., Klopfer, D. & Wang, Y. (1988). Expertise in a complex skill: Diagnosing X-ray pictures. In M. T. H. Chi, R. Glaser & M. J. Farr (Eds.), The nature of expertise. Lawrence Erlbaum Associates, Hillsdale, NJ.

Lewis, C., (1978). Production system model of practice effects. Dissertation, University of Michigan.

Lewis, C., (1981). Skill in algebra. In J. R. Anderson (Ed.), Cognitive skills and their acqusition. Lawrence Erlbaum, Hillsdale, NJ.

Lewis, C. and Olson, G., (1987). Can principles of cognition lower the barriers to programming? In G. M. Olson, S. Sheppard and E. Soloway (Eds.), Empirical studies of programmers: second workshop, Ablex, Norwood, NJ.

Leventhal, L. M., (1987). Discourse rules in program comprehension: emergence of construct affordances rule, In Proc. of INTERACT'87, H. J. Bullinger and B. Shackel (Eds.), Elsevier, North - Holland.

Leventhal, L. M., (1988). Experience of programming beauty: some patterns of programming aesthetics. *Int. J. Man - Machine Studies*, **28**, 525 - 550.

Littman, D. C. and Soloway, E., (1988). Evaluating ITS's: The cognitive science persepctive. In M. C. Polson and J. J. Richardson (Eds.), Foundations of intelligent tutoring systems. Lawrence Erlbaum, Hillsdale, NJ.

Logan, G. D., (1979). On the use of concurrent memory load to measure attention and automaticity. *Journal of Experimental Psychology. Human Perception and Performance,* 5, 189-207.

Maiden, N. and Sutcliffe, A., (1991). The abuse of reuse: Why software reuse must be taken into care. Paper presented at HCI'91. Edinburgh

Mandler, J. M. and Johnson, N. S., (1977). Remembrance of things parsed: Story structure and recall. *Cognitive Psychology*, **9**, 111 - 151.

Mayer, R. E., (1976). Comprehension as affected by structure of problem represention. *Memory and Cognition*, **4** (3), 249 - 255.

Mayer, R. E., (1985). Learning in complex domains: a cognitive analysis of computer programming. In the psychology of learning and motivation, G. H. Bower (Ed.), Vol 19, Academic Press.

Mayer, R. E., (1987). Cognitive aspects of learning and using a programming language. In Interfacing thought: Cognitive aspects of HCI, J. M. Carroll (Ed.), MIT Press.

Mayes, J. T., Draper, S. W., McGregor, M. A. and Oatley, K., (1988). Information flow in a user interface: the effects of experience and context on the recall of MacWrite screens. In D. M. Jones and R. Winder (Eds.), People and Computers IV. Cambridge University Press, Cambridge.

McCalla, G. I. and Reid, L., (1982). Plan creation, plan execution and knowldge acquisition in a dynamic microworld. International Journal of Man-Machine Studies, 16, 89 - 112.

McClelland, J. L. and Rumelhart, D. E., (1986). Parallel distributed processing. Vol. 2 - Psychological and Biological models. MIT Press, Cambridge, MA.

McKeithen, K. B., Reitman, J. S., Rueter, H. H. and Hirtle, S. C., (1981). Knowledge organization and skill differences in computer programs. *Cognitive Psychology*, **13**, 307-325.

McKendree, J. and Anderson, J. R., (1987). Effect of Practice on Knowledge and use of Basic Lisp. In Interfacing thought: Cognitive aspects of HCI, J. M. Carroll (Ed.), MIT Press.

McKoon, G. and Ratcliff, R., (1984). Priming and on-line text comprehension. In D. E. Kieras and M. A. Just (Eds.), New methods in reading comprehension research. Lawrence Elbaum, Hillsdale, NJ.

Meyer, B. J. F., (1975). The organization of prose and its effect on memory. North-Holland, Amsterdam.

Miara, R. J., Musselman, J. A, Navarro, J. A. and Shneiderman., (1983). Program Indentation and Comprehensibility. *Comms ACM*, **26** (11), 861 - 867.

Micallef, J., (1988). Encapsulation, reusability and extensibility in object-oriented programming languages. *Journal of Object-Oriented Programming*, **1**, 1, 12 - 38.

Miller, G. A., (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, **63**, 81 - 93.

Miller, G. A., Galanter, E. and Pribram, K., (1960). Plans and the structure of behaviour. Holt, Rinehart and Wilson, New York.

Miller, L., (1978). Has artificial intelligence contributed to an understanding of the human mind? A critique of arguments for and against. *Cognitive Science*, **2**(2), 111 - 128.

Morris, D., Theaker, C. J., Phillips, R., & Love, W. (1988). Human-Computer Interface Recording. *The Computer Journal*, **31**, 5, 437 - 444.

Myers, G. J., (1978). A controlled experiment in program testing and code walkthroughs/inspection. *Communications of the ACM*, **21**, 760 -768.

Mynatt, B. T., (1984). The effects of semantic complexity on the comprehension of program modules. *Int. J. Man - Machine Studies*, **21**, 91 - 103.

Neal, L. R., (1987a). Cognition-sensitive design and user modeling for syntax-directed editors. In J. M. Carroll and P. Tanner (Eds.), Human Factors in Computing Systems IV, Nort-Holland, Amsterdam.

Neal, L. R., (1987b). User modeling for syntax-directed editors. In H.-J. Bullinger and B. Shackel (Eds.), Proceedings of Interact'87. North-Holland, Amserdam.

Neisser, U., (1976). Cognition and Reality. Principles and implications of cognitive psychology. W. H. Freeman, San Francisco.

Neves, D. M. and Anderson, J. R., (1981). Knowledge compilation: mechanisms for the automatization of cognitive skills. In J. R. Anderson (Ed.), Cognitive skills and their acquisition, Lawrence Erlbaum, Hillsdale, NJ.

Newell, A., (1969). Heuristic programming: Ill-structured problems. In J. Aronofsky (Ed.), Progress in operations research, Wiley.

Newell, A., (1973). You can't play 20 questions with nature and win. In W. G. Chase (Ed.), Visual information processing. Academic Press, New York.

Newell, A. and Rosenbloom, P. S., (1981). Mechanisms of skill acquisition and the law of practice. In J. R. Anderson (Ed.), Cognitive skills and their acquisition. Erlbaum, Hillsdale, NJ.

Newell, A., Shaw, J. C. and Simon, H., (1958). Elements of a theory of human problem solving. *Psychological Review*, **65**, 151 166.

Newell, A. and Simon, H., (1972). Human problem solving. Prentice Hall, Englewood Cliffs, NJ.

Norman, D. A., (1981). Caterorisation of action slips. *Psychological Review*, **88**, 1-15.

Norman, D. A., (1983). Some observations on mental models. In D. Gentner and A. L. Stevens (Eds.), Mental Models. Lawrence Erlbaum, Hillsdale, NJ.

Norman, D., (1986). Cognitive Engineering. In D. A. Norman and S. W. Draper (Eds.), User-Centered System Design. Lawrence Erlbaum, Hillsdale, NJ.

Norman, D. A., (1988). The psychology of everyday things. Basic Books, New York.

Ormerod, T. C., (1991). What really determines the success of novices learning programming - A collection of protocols. PPIG'91.

Ormerod, T. C., Manktelow, K. I., Robson, E. H. and Steward, A. P., (1986). Content and representation effects with reasoning tasks in PROLOG form. *Behaviour and Information Technology*, **5** (2), 157 - 168.

Ormerod, T. C., Manktelow, K. I., Steward, A. P. and Robson, E. H., (1990). The effects of content and representation on the transfer of PROLOG reasoning skills. In K. J. Gilhooly, M. T. G. Keane, R. H. Logie and G. Erdos, eds., Lines of Thinking: Reflections on the Psychology of Thought ,Vol 1, Wiley.

Parker, J. and Hendley, B., (1987). The UNIVERSE program development environment. In Proc. of INTERACT'87. H. J. Bullinger and B. Shackel (Eds.), Elsevier, North - Holland.

Parker, J. and Hendley, B., (1988). The re-use of low-level programming knowledge in the UNIVERSE programming environment. In P. Brereton (Ed.), Software Engineering Environments, Ellis Horwood, Chichester, 1988.

Pavard, B., (1985). La conception des systemes de traitement de texte. *Intellectica* , **1**, 37-68.

Payne, S. J., (1987). Complex problem spaces: Understanding the knowledge needed to use interactive systems. In H. Bullinger and B. Shackel (Eds.), Human Computer Interaction - INTERACT'87. North-Holland, Amsterdam.

Payne, S. J., (1988). Methods and mental models in theories of cognitive skill. In J. Self (Ed.), Artificial Intelligence and Human Learning. Chapman and Hall, London.

Payne, S. J., (1990). Looking HCI in the I. In D. Diaper, D. Gilmore, G, Cockton and B, Shackel (Eds.) Human Computer Interaction - INTERACT'90., North-Holland, Amsterdam.

Payne, S. J., (in press). Display-based action at the user interface. *International Journal of Man-Machine Studies.*

Payne, S. J., Squibb, H. R. and Howes, A., (1990). The nature of device models: The yoked state space hypothesis and some experiments with text editors. *Human-Computer Interaction*, **5**, 415 - 444.

Pennington, N., (1985). Cognitve components of expertise in computer programming: A review of the literature. *Psychological Documents*, **15**, 2702.

Pennington, N., (1987 a). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, **19**, 295 - 341.

Pennington, N., (1987 b). Comprehension strategies in programming. In G. M. Olson, S. Sheppard and E. Soloway (Eds.), Empirical studies of programmers: second workshop, Ablex, Norwood, NJ.

Pennington, N. and Grabowski, B., (1990). The tasks of programming. In J.-M. Hoc, T. R. G. Green, R. Samurcay and D. J. Gilmore (Eds.), Psychology of Programming, London, Academic Press.

Perkins, D. N. and Martin, F., (1986). Fragile knowledge and neglected strategies in novice programmers. Empirical studies of programmers, First Workshop, Ablex, Norwood, NJ.

Petre, M. and Winder, R., (1988). Issues governing the suitability of programming languages for programming tasks. In People and Computers IV, Proc. HCI'88, UMIST. (Eds.) D. Jones and R. Winder, Cambridge University Press, Cambridge.

Pirolli, P. L., (1986). A cognitive model and computer tutor for programming recursion. *Human-Computer Interaction,* **2**, 319 - 355.

Pirolli, P. L. and Anderson, J. R., (1985). The role of learning from examples in the acquisition of recursive programming skills, *Canadian Journal of Psychology*, **39 (2)**, 240-272.

Pirolli, P. and Bielaczyc, K., (1989). Empirical analysis of self-explanation and transfer in learning to program. In the proceedings of the Eleventh conference of the Cognitive Science Society. Ann Arbor, Michigan. Lawrence Erlbaum, Hillsdale, NJ. 405 - 457.

Polya, G., (1973). How to solve it. Princeton University Press, Princeton, NJ.

Rajan, T., (1986). A principled design for an animated view of program execution for novice programmers. Open University HCRL Research Report Number 19a. December 1986.

Ranney, M. and Reiser B. J., (1989). Reasoning and explanation in an intelligent tutor for programming. In G. Salvendy and M. J. Smith (Eds.), Designing and using Human-Computer Interfaces and Knowledge based systems. Elsevier, Amsterdam.

Ratcliff, B. and Siddiqi, J. I. A., (1985). An empirical investigation into problem decomposition strategies used in program design. *Int. J. Man - Machine Studies*, **22**, 77 - 90.

Reason, J. T., (1979). Actions as not planned: the price of automatisation. In G. Underwood and R. Stevens (Eds.), Aspects of consciousness. Academic Press, London.

Reif, R. and Heller, J. I., (1982). Knowledge structure and problem solving in physics. *Educational Psychologist*, **17**, 102 - 127.

Reitman, J., (1976). Skilled perception in Go: Deducing memory structures from inter-response times. *Cognitive Psychology*, **8**, 336 - 356.

Reitman, J. and Rueter, H., (1980). Organization Revealed by Recall and confirmed by Pauses, *Cognitive Psychology*, **12**, 554 - 581.

Reitman, W., (1965). Cognition and thought, Wiley, New York.

Rich, C., (1981). A formal representation for plans in the programmer's apprentice. In Proceedings of IJCAI-81, Vancouver, BC, 1044 - 1052.

Rich, C., Shrobe, H. E. and Waters, R. C., (1979). An overview of the programmers apprentice. In Proceedings of IJCAI-79, Tokto, Japan.

Rich, C. and Wills, L. M., (1990). Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 82 - 89.

Rist, R. S., (1986a). Plans in programming: definition, demonstration, and development. Empirical Studies of Programmers, First Workshop, Ablex, Norwood, NJ.

Rist, R. S., (1986b). Focus and learning in program design. Proceedings of the 8th. Cognitive Science conference, Amherst, MA.

Rist, R. S., (1986). Learning by doing: description of model. Yale Research Report.

Rist, R. S., (1989). Schema creation in programming, *Cognitive Science*, **13**, 389-414.

Rist, R. S., (1990). Variability in program design: the interaction of process with knowledge, *International Journal of Man-Machine Studies*, **33**, 305-322

Rist, R. S., and Bevemyr, J., (1989). PARE: A cognitively base programmer analyser. 12th Int Conf. on AI.

Robertson, S. P., & Black, J. B. (1986). Planning in text editing behavior. In A. Jonda (Ed.), Human factors in computing systems, North-Holland, New York.

Robertson, S. P. and Chiung-Chen, Y., (1990). Common cognitive representations of program code across tasks and languages. *International Journal of Man-Machine Studies*, **33**, 343 - 360.

Robertson, S. P., Davis, E. F., Okabe, K. and Fitz-Randolf, D., (1990). Program comprehension beyond the line. Proceedings of INTERACT'90. D. Diaper, D. Gilmore, G. Cockton and B. Shackel, (Eds.), Elsevier, North-Holland, Amsterdam.

Roske-Hofstrand, R. J. and Paap, K. R., (1986). Cognitive networks as a guide to menu organization: and application in the automated cockpit. *Ergonomics*, **29**, 1301 - 1312.

Rosson, M. B. and Alpert, S. R., (1990). The cognitive consequences of object-oriented design. *Human-Computer Interaction*, **5**, 345 - 379.

Rumelhart, D. E. (1975). Notes on a schema for stories. In D. G. Bobrow and A. Collins (eds.), Representation and understanding: Studies in cognitive science. Academic Press, New York.

Rumelhart, D. E., (1977). Understanding and summarizing brief stories. In D. Laberge and S. J. Samuels (Eds.), Basic processes in reading. Lawrence Erlbaum, Hillsdale, NJ.

Rumelhart, D. E., (1980). Schemata: The basic building blocks of cognition. In R. Spiro, B. Bruce and W. Brewer (Eds.), Theoretical issues in reading comprehension. Lawrence Erlbaum, Hillsdale, NJ.

Rumelhart, D. E.and McClelland, J. L., (1986). Parallel distributed processing. Vol. 1 - Foundations. MIT Press, Cambridge, MA.

Rumelhart, D. E. and Norman, D. A., (1978). Accretion, Tuning and Restructuring: Three modes of learning. In J. W. Cotton and R. Klatsky, (Eds.), Semantic factors in cognition, Lawrence Erlbaum, Hillsdale, NJ.

Rumelhart, D. E. and Norman, D. A., (1981). Analogical processes in learning. In J. R. Anderson (Ed.), Cognitive skills and their acquisition. Lawrence Erlbaum, Hillsdale, NJ.

Sacerdoti, E. D., (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, **5**, 115 - 135.

Sacerdoti, E. D., (1977). A structure for plans and behavior. Elsevier, North-Holland.

Sajaniemi, J. and Nienelainen A. (1989). Program editing based on variable plans: A cognitive approach to program manipulation. In G. Salvendy and M. J. Smith (Eds.), Designing and using Human-Computer Interfaces and Knowledge based systems. Elsevier, Amsterdam.

Schank, R. C., (1979). Intrestingness: Controlling inferences. *Artificial Intelligence*, **12**, 273 - 297.

Schank, R. C., (1980). Language and Memory. *Cognitive Science*, **4**, 243 - 284.

Schank, R. C., (1981). Reading and Understanding. Lawrence Erlbaum, Hillsdale, NJ.

Schank, R. C. and Abelson, R., (1977). Scripts, Plans, Goals and Understanding. Lawrence Erlbaum, Hillsdale, NJ.

Scheider, W. and Detweiler, M., (1987). A connectionist/control architecture for working memory. In G. H. Bower (Ed.), The psychology of learning and motivation, Vol 21, Academic Press, New York.

Schoenfeld, A. H. (1985). Mathematical problem solving. Academic press, London.

Schoenfeld, A. and Herrmann, D., (1982). Problem perception and knowledge structures in expert and novice mathematical problem solvers. *Journal of Experimental Psychology: Learning, Memory and Cognition*, **8**, 484 - 494.

Scholtz, J. and Wiedenbeck, S., (1989). Learning a new programming language. In G. Salvendy and M. J. Smith (Eds.), Designing and using Human-Computer Interfaces and Knowledge based systems. Elsevier, Amsterdam.

Schönflug, W., (1986a). External information storage: An issue for the psychology of memory. In F. Klix and H. Hagendorf (eds.), Human memory and cognitive capabilities; mechanisms and performance. Elsevier Science Publishers.

Schönflug, W., (1986b). The trade-off between internal and external information storage. *Journal of Memory and Language*, **25**, 657-675.

Schvaneveldt, R. W., Durso, F. T. and Dearholt, D. W., (1985). Pathfinder: Scaling with network structures. Memorandum in Computer and Cognitive Science, MCCS-85-9, Computing Research Laboratory, New Mexico State University.

Sharples, M and O'Mally, C., (1988). A framework for the design of a writer's assistant. In J. Self (Ed.), Artificial Intelligence and Human Learning. Chapman and Hall, London.

Shneiderman, B., (1976). Exploratory experiments in programmer behaviour. *International Journal of Computer and Information Science*, **5**, 123 - 143.

Shneiderman, B., (1977). Measuring computer program quality and comprehension. *International Journal of Man-Machine Studies*, **9**, 465 - 478.

Shneiderman, B., (1980). Software psychology: Human factors in computer and information systems. Little, Brown, Boston, MA.

Shneiderman, B. and Mayer, R., (1979). Syntactic/Semantic Interactions in Programmer Behavior: A model and Experimental Results. *International Journal of Computer and Information Sciences*, **8**, 3, 219 - 238.

Shneiderman, B., Shafer, P., Simon, R. and Weldon, L., (1986). Display Strategies for Program Browsing: Concepts and Experiment. *IEEE Software*, **3**, 3, 7 - 15.

Siddiqi, J. I. A., (1985). A model of programmer designer behaviour. In people and computers: designing the interface, P. Johnson and S. Cook (Eds.), Cambridge University Press.

Siddiqi, J. I. A. and Ratcliff, B. (1989). Specification influences in program design. *Int. J. Man-Machine Studies*, **31**, 393-404.

Siddiqi, J. I. A. and Sumiga, J. H., (1989). Empirical evaluation of a proposed model of the program design process. In P. Zundo and P. Agrawal (Eds.), Proc. 4th. Symposium on empirical foundations of information and software sciences (1987). Plenum Press, 1989. 333 - 345.

Sime, M. E., Arblaster, A. T. and Green, T. R. G., (1977). Structuring the programmer's task. *J. Occup. Psychology*, **50**, 205 - 216.

Sime, M. E., Green, T. R. G. and Guest, D. J., (1973). Psychological evaluation of two conditional constructions used in computer languages. *International Journal of Man-Machine Studies*, **5**, 123 - 143.

Sime, M. E., Green, T. R. G. and Guest, D. J., (1977).Scope marking in computer conditionals - A psychological evaluation. *International Journal of Man-Machine Studies*, **9**, 107 - 118.

Simon, H. A., (1962). The arcitecture of complexity. *Proc. Amer. Philosoph. Soc.*, **106**, 467 - 482.

Simon, H., (1973). The structure of ill-structured problems. *Artificial Intelligence*, **4**, 181 - 201.

Simon, H. A., (1975). The functional equivalence of problem solving skills, *Cognitive Psychology*, **7**, 268 - 288.

Simon, H. A., (1978). Information-processing theory of human problem solving. In W. K. Estes (Ed.), Handbook of learning and cognitive processes: Vol 5. Lawrence Erlbaum, Hillsdale, NJ.

Simon, H. and Reed, S. K., (1976) Modelling strategy shifts on a problem solving task. *Cognitive Psychology*, **8**, 86 - 97.

Singley, M. K. and Anderson, J. R., (1985). The transfer of text-editing skill. *International Journal of Man-Machine Studies*, **22**, 403 - 423.

Singley, M. K. and Anderson, J. R., (1988). A keystroke analysis of learning and transfer in text editing. *Human-Computer Interaction*, **3**, 223 -274.

Smith, E. E. and Goodman, L., (1984). Understanding instructions: the role of an explanatory schema. *Cognition and Instruction*, **1**, 359 - 396.

Soloway, E., Bonar, J. and Ehrlich, K., (1983). Cognitive strategies and looping constructs: an empirical study. *Comms ACM*, **26** (11), 853 - 860.

Soloway, E. and Ehrlich, K., (1984). Empirical studies of programming knowledge. *IEEE Trans. SE*, **SE - 10**(5), 595 -609.

Soloway, E., Ehrlich, K. and Black, J. B., (1983). Beyond numbers : Don't ask "how many"...Ask "Why". In A. Janda (Ed.), Proc. CHI'83. Boston, MA. North-Holland.

Soloway, E., Ehrlich, K., Bonar, J and Greenspan, J., (1982). What do novices know about programming. In A Badre and B. Shneiderman (Eds.), Directions in Human-Computer Interaction, Ablex, Norwood, NJ.

Soloway, E., Ehrlich, K. and Gold, E., (1983). Reading a program is like reading a story (well, Almost). Proc. 5th. Annual Conf. of the Cognitive Science Society. Rochester, NY.

Spohrer, J. C. and Soloway, E., (1986). Alternatives to construct-based programming misconceptions. In Human factors in computing systems, Proc. CHI'86, North - Holland, Amsterdam.

Spohrer, J. C., Soloway, E. and Pope, E., (1985). Where the bugs are. In Human factors in computing systems 2, Proc. CHI'85, L. Borman and B. Curtis (Eds.), North - Holland, Amsterdam.

Staszewski, J. J., (1988). Skilled memory and expert mental calculation. In Chi, M. T. H., Glaser, R. and Farr, M. J., (Eds.), The nature of expertise, Lawrence Erlbaum, Hillsdale, NJ.

Stone, D. N., Jordan, E. W. and Wright, M. K., (1990). The impact of Pascal education on debugging skill. *Int. J. Man-Machine Studies*, **33**, 81 - 95.

Strauss, S. and Stavy, R., (1982). U-shaped behavioral growth: implications for theories of development. In W. W. Hartup (Ed.), Review of child devlopment research. University of Chicago Press, Chicago.

Sutcliffe, A. and Maiden, N., (1990). Software reusability: Delivering productivity gains or short cuts. In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.), Proceedings of INTERACT'90. North-Holland, Amsterdam.

Thomas, J. C. and Carroll, J. M., (1979). The psychological study of design. *Design Studies.* **1**, 5-11.

Thomas, M. and Zweben, S., (1986). The effects of program-dependent and program-independent deletions on software cloze tests. Empirical Studies of Programmers, First Workshop, Ablex, Norwood, NJ.

Trabasso, T., Secco, T. and van den Broeck., (1982). Causal cohesion and story coherence. In H. Mandl, N. L. Stien and T. Trabasso (Eds.), Learning and comprehension of text. Lawrence Erlbaum, Hillsdale, NJ.

Tversky, A. and Kahneman, D., (1973). Availability: A heuristic for judging frequency and probability. *Cognitive Psychology*, **5**, 207 - 232.

van Dijk, T. and Kintsch, W., (1983). Strategies of discourse comprehension. Academic Press, New York, NY.

Van Laar, D., (1989). Evaluating a colour coding programming support tool. In A. Sutcliffe and L. Macaular (Eds.), People and Computers V. Cambridge University Press, Cambridge.

VanLehn, K., (1982). Bugs are not enough: emprical studies of bugs, impasses and repairs in procedural skills. *Journal of Mathematical Behaviour*, **3**, 3 - 72.

VanLehn, K., (1990). Mind Bugs: The origins of procedural misconceptions. MIT Press, Cambridge, MA.

van Merrienboer, J. J. G., (1990). What cognitive science may learn from instructional design: A case study in introductory computer programming. AERA Annual meeting, Boston, USA.

Vessey, I., (1985). Expertise in debugging computer programs: A process analysis. *Int. J. Man-Machine Studies,* **23**, 459 - 494.

Vessey, I., (1986). Expertise in debugging computer programs: An analysis of the contents of verbal protocals, *IEEE trans Systems, Man and Cybernetics*, **SMC - 16** (5), 621 - 637.

Vessey, I., (1989). Toward a theory of computer program bugs: an empirical test. *Int. J. Man-Machine Studies*, **30**, 23 - 46.

Vessey, I. and Weber, R., (1984a). Conditional statements and program coding: an experimental evaluation. *Int. J. Man-Machine Studies,* **21**, 161 - 190.

Vessey, I. and Weber, R., (1984b). Research on Structured Programming: An Empiricist's Evaluation. *IEEE Trans. Soft. Eng.*, **SE-10**, 4, 397 - 407.

Visser, W., (1987). Strategies in programming programmable controllers: a field study on a professional programmer. In G. M. Olson, S. Sheppard and E. Soloway (Eds.), Empirical studies of programmers: second workshop, Ablex, Norwood, NJ.

Visser, W. and Hoc, J-M., (1989). Program design strategies. To appear in D. Gilmore, T. R. G. Green, J-M. Hoc and R. Samurcay (Eds.), Psychology of programming, Academic Press, London.

Visser, W. and Morais, A., (1991). Concurrent use of different expertise elicitation methods applied to the study of the programming activity. In M. J. Tauber and D. Ackermann (Eds.), Mental Models in HCI, Elsevier, North-Holland.

Waddington, R. and Henry, R., (1989). The effect on program authorship on novice debugging performance. In G. Salvendy and M. J. Smith (Eds.), Designing and using Human-Computer Interfaces and Knowledge based systems. Elsevier, Amsterdam.

Waddington, R and Henry, R., (1991) Selective Information hiding: A debugging Technique to address some of the problems of Novice programmers. Proc. Int Conference on HCI, Suttgart, 1991.

Waters, R. C., (1979). A method for analysing loop programs. *IEEE Transactions on Software Engineering*, **5**, 237 - 247.

Weber, G., (1991). Explanation Based retrieval in a case-based learning model. Proc of 13th meeting of the cognitive science society. Chicago.

Weber, G., (in press a). Episodic Modelling in an intelligent tutoring system. D. G. Bouwhuis (Ed.), Cognitive modelling and Interactive Environments, NATO ASI Series F. Springer, Berlin.

Weber, G., (in press b). An Episodic Student model for an Intelligent LISP Tutor, D. H. Joanssen and T. Mayes (Eds.), Mindtools: Cognitive technologies for modelling knowledge. Springer, Berlin.

Weber, G and Mollenberg, A., (Date unknown). STRUEDI: A tutoring system for LISP Beginners, Draft Repoart

Weiser, M. and Shertz, J., (1983). Programming problem representation in novice and expert programmers. *Int. J. Man - Machine Studies*, **19**, 391 - 398.

Weissman, L. M., (1974). Psychological complexity of computer programs: an experimental methodology. *Sigplan notes*, **9** (6), 25 - 36.

White, R., (1988). Effects of Pascal upon the learning of Prolog - a preliminary study. Presented at the International Conference on Thinking, Aberdeen, August 1988.

Whitefield, A. D., (1985). Constructing and applying a model of the user for computer systems development. PhD Thesis, University College, London

Widowski, D. and Eyferth, K., (1986). Comprehending and recalling computer programs of different structural and semantic complexity by experts and novices. In H-P Willumeit (Ed). Human Decision making and manual control. Elsevier, North-Holland.

Wiedenbeck, S., (1985). Novice/expert differences in programming skills. *Int. J. Man-Machine Studies*, **23**, 383 - 390.

Wiedenbeck, S., (1986a). Processes in computer program comprehension. Empirical Studies of Programmers, First Workshop, Ablex, Norwood, NJ.

Wiedenbeck, S., (1986b). Beacons in computer program comprehension. *Int. J. Man - Machine Studies*, 25 , 697-709.

Wiedenbeck, S., (1989). Learning iteration and recursion from examples. *Int. J. Man - Machine Studies*, 30 ,1 - 22.

Wiedenbeck, S. and Scholtz, J., (1989). Beacons: A knowledge structure in program comprehension. In G. Salvendy and M. J. Smith (Eds.), Designing and using Human-Computer Interfaces and Knowledge based systems. Elsevier, Amsterdam.

Wills, L. M., (1990). Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45, 113 - 171.

Winograd, T., (1974). Natural language understanding. Academic Press, New York.

Winograd, T. and Flores, F., (1987). Understanding computers and cognition: A new foundation for design. Addision Wesley, Reading, MA.

Wirth, N., (1971). Program development by stepwise refinement, *Communications of the ACM*, 14, 4.

Woodfield, S. N., Dunsmore, H. E. and Shen, V. Y., (1981).The effects of modularization and comments on program comprehension. Proc. 5th. Int. Conf. on Software Engineering. IEEE press, 215 - 223.

Young, R. M., (1979). Production systems for modelling human cognition. In D. Michie (Ed.), Expert systems in the microelectronic age. Edinburgh University Press, Edinburgh.

Young, R. M., Howes, A. and Wittington, J., (1990). A knowledge analysis of interactivity. In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.), Human-Computer Interaction, Interact'90. North-Holland, Amsterdam.

Young, R. M. and Simon, T., (1987). Planning in the context of human-computer interaction. In D. Diaper and R. Winder (Eds.), People and Computers 3. Cambridge University Press, Cambridge.

Youngs, E. A., (1974). Human Errors in Programming. *Int. J. of Man-Machine Studies*, 6, 361-376.

Yourdon, E., (1975). Techniques of program structure and design. Prentice-Hall, Englewood Cliffs, NJ.

Yu, C. and Robertson, S. P., (1988). Plan-based representations of Pascal and Fortran code. In Proc. CHI'88

Zeitz, C. M. and Spoeher, K. T., (1989). Knowledge organisation and the acqusition of procedural expertise. *Applied cognitive psychology*, 3, 313 - 336.