Focal Structures in Prolog

Pablo Romero
School of Cognitive and Computing Sciences
Sussex University, U.K.
email: juanr@cogs.susx.ac.uk *

Abstract

Several studies have suggested that the mental structures of programmers of procedural languages have a close relationship with a model of structural knowledge known as Programming Plans. However, it is not clear that this is the case for Prolog, specially because this language has important differences when compared to procedural languages. It does not have obvious syntactic cues to mark blocks of code (begin/end, repeat/until, etc). Also, its powerful primitives (unification and backtracking) and the extensive use of recursion might influence the way programmers comprehend Prolog code in a significant way.

This paper reports an experiment that tries to characterise the nature of the mental models that programmers build when comprehending Prolog code by finding out which of several structural models, Programming Plans among them, are most relevant for the case of this programming language. The results suggest that Prolog Schemas, a construct related to data structure relationships, is of central importance to Prolog programmers. This result contrasts with those obtained for procedural languages, where Programming Plans, a concept related to functional information, seems to be the dominant model.

Keywords: psychology of programming, program comprehension, Prolog, Prolog Schemas.

1 Introduction

Program comprehension is a skill that is central to programming, so having a clear picture of comprehension as a cognitive process is a prerequisite to building models of programming tasks such as debugging, modification, reuse, etc. Yet comprehension has been studied mainly for languages belonging to the structured programming paradigm.

Studying programming languages very different from the structured paradigm can offer interesting ways to test the findings of the area. The Programming Plan concept has been the dominant model used to represent the structural knowledge comprised in computer programs (Detienne, 1990). It has been suggested that this model has a close relationship to the mental models programmers build when they perform program comprehension. However, these claims have yet to be tested for programming paradigms other than the structured one. A programming language significantly different from this main trend is Prolog. Prolog is in a class of its own because of its declarative nature and its powerful primitives.

The nature and characteristics of the mental models built during comprehension for the case of Prolog are not clear. Several studies have tried, without much success, to find evidence of a relationship between the Programming Plans idea and the mental models of Prolog programmers (Bellamy &

*Supported by a grant from the Mexican Council for Science and Technology, CONACYT

Goal:	find an occurrence of ?x	
	CODE	PLAN TERMS
Plan:	?found := false	initialise to not found
	loop through category of ?x	
	if ?x then	
	?found := true	set it to true
	... := ?x	use it

Figure 1: Plan description
From Gilmore and Green (1988)

Gilmore, 1990; Ormerod & Ball, 1993). The experiment described in this paper tries to characterise certain aspects of these mental models by looking at several structural models proposed for the case of Prolog (Programming Plans among them).

This document is divided into three sections. The next section gives a brief account of program comprehension studies, the following part describes the experiment and the final section analyses its results and compares them to those reported for other programming languages.

2 Program comprehension

Program comprehension is a complex cognitive process that involves the acknowledgement and understanding of several elements. First of all, the result of this process is a *mental model* that the programmer builds of the program she has studied. The qualities of this mental model vary according to several factors, among them the programmer's skill level, the size of the program, the task in hand, etc.

In order to understand the sources of knowledge that programmers use to build these comprehension mental models, several *structural models* of this programming knowledge have been proposed. One of the most successful of such models is the idea of Programming Plans.

These models propose specific structures that are said to be a good approximation of the internal knowledge structures that enable programmers to organise programs in a particular way. This structural organisation sometimes highlights a specific *aspect* of the code. Pennington (1987) identified structural models with the term Programming Knowledge and the code aspects with the notion of Text Abstractions.

2.1 The comprehension process

One of the earliest theories of program comprehension was proposed by Brooks (1983). Brooks proposed a theoretical framework to understand behavioural differences in program comprehension. He regards comprehension as a process of domain reconstruction. This reconstruction involves establishing mappings from the problem domain to the program domain via some other intermediate domains. This process of establishing mappings consists of generating and refining hypotheses about the executing program and its relation to other domains. Hypothesis refinement is performed in a top-down fashion. This process begins with a primary, top-level hypothesis which is decomposed into several subsidiary, more specific hypotheses. The generation of hypotheses is performed by retrieving structural units from the programmers knowledge. These structural knowledge units are used to generate more hypotheses or are matched against the program's code. As a result of this matching process, the code is organised into meaningful chunks or units. These chunks can be considered as the external analogues of the

```
p(X):-
    g(X,Y),
    p(Y).
```

Figure 2: The *before* technique

From Bowles and Brna (1993)

```
length([],L,L).
length([H|T],L0,L):-
                    L1 is L0+1,
                    length(T,L1,L).
```

Figure 3: An occurrence of the *before* technique

From Bowles and Brna (1993)

programmer's structural knowledge. According to Brooks, in order to perform this organisation of the program into meaningful chunks, programmers look for specific patterns of code which can confirm the proposed hypothesis. These patterns of code are known as *key segments* of the code's meaningful chunks.

The next section describes several models proposed to explain the nature and characteristics of the programmers' internal structural knowledge used in the comprehension process.

2.2 Structural models

A distinction has to be made between a structural model and the organisation of a specific program according to the application of a structural model. A structural model is a construct that is used to explain some aspects of the programming knowledge possessed by programmers, particularly by experienced programmers. Some segments of the program code are more relevant than others for a specific structural model. Therefore, these segments of code can be considered as specific instances of their associated structural model. I will refer to these segments of code as the structural model's *instances.*

Several studies have proposed that a concept known as 'Programming Plans' (Pennington, 1987; Gilmore & Green, 1988; Davies, 1990) can be used to explain some aspects of the programmers' structural knowledge. These studies suggest that there is a strong link between the mental model that a programmer builds and an organisation of the code according to a Plan-like structure. Plans are proposed as the external analogue of the programmers' internal structural knowledge that is used to organise the program as a hierarchy of meaningful units. These units are considered as frames that comprise stereotypical programming procedures and whose slots can be filled with variables related to the specific problem being solved. In this way, Plans can be seen as Data Structures that represent generic concepts stored in memory. Figure 1 gives an example of a plan instance.

Studies of Programming Plans have considered mainly procedural languages. Some studies have tried, without much success, to find evidence of a relationship between Plans and Prolog programmers' mental models (Bellamy & Gilmore, 1990; Ormerod & Ball, 1993).

There are alternative structural models for Prolog. Brna, Bundy, Todd, Eisenstadt, Looi, and Pain (1991) and Bowles and Brna (1993) propose that Prolog programmers' structural knowledge is related to 'Prolog Techniques'. This structural model is similar to Plans, but it comprises knowledge about how to perform specific Prolog operations. An instance of a basic programming technique is given in figure 2. This technique's instance is called the *before* technique because the value of Y is constructed in the subgoal *g* and then sent to the recursive call. Figure 3 illustrates an occurrence of this instance in the predicate *length/3.*

Another structural model for Prolog is 'Prolog Schemas'. Gegg-Harrison (1991) proposes this structural

```
schema_C([E|T],E,<< &1 >>).
schema_C([H|T],E,<< &2 >>:-
                        <E\=H>,
                        <pre_pred(<< &3 >>,H,<< &4 >>)>,
                        schema_C(T,E,<< &5 >>),
                        <post_pred(<< &6 >>,H,<< &7 >>)>,
```

Figure 4: An example of a simple Prolog schema

From Gegg-Harrison (1991)

model and describes a set of common Prolog Schema instances for list processing. A specific example from this set is given in figure 4. In this example, << &n >> denotes any number of Prolog arguments, and clauses surrounded by <> are optional. This example deals with the task of processing a list until the first occurrence of an element is found. The base case ensures that E, the element that is being searched for, is found. The second clause optionally checks that the list element being processed is not the one which is being looked for, performs an optional process, makes a recursive call trying to find the element in the tail of the list and calls a second optional process. This schema instance is very similar to the example of a programming plan instance given in Figure 1.

Techniques and Schemas as structural models for Prolog were proposed for teaching purposes. Their authors do not claim a relationship between these constructs and Prolog programmers mental models. One of the purposes of this paper is to explore whether such a relationship exists.

There has been some research about the notion of key segments in Programming Plans (Wiedenbeck, 1986; Rist, 1989; Davies, 1994; Rist, 1995). This indicates that Plans are not monolithic structures but that there are elements of Plans that are more relevant than others. These key elements represent the central or focal action of a Plan. For example, if the programming task is to compute an average, the key element of it will be the place where the division between the running total and the number of items takes place (Rist, 1995). Wiedenbeck (1986) gives empirical evidence that supports this notion of key elements. In her study, novices and experienced programmers tried to understand and memorise a short Pascal program. After this study period, they were asked to recall as much as they could of the program code. The results showed that experienced programmers, unlike novices, recalled key segments much better than other parts of the code.

There has not been any research of this kind for the case of Prolog Techniques and Prolog Schemas yet, but the definition of Schemas includes the notion of compulsory and optional elements inside these structures. The compulsory elements could be considered as the key segments for Prolog Schemas.

2.3 Code aspects

A structural organisation sometimes highlights a specific aspect of the code. Code aspects refer to the different ways in which a program can be interpreted, or in Pennington's words, to the different kinds of information implicit in the program text. Some of these different kinds of information can be Function, Data Structure, Data-flow and Control-flow. Function refers to what the program does, Data Structure to the programming language objects that are used in order to implement a solution to the programming problem. Data-flow refers to how these objects are related in the program and Control-flow to the sequence of actions that will occur when the program is executed (Pennington, 1987).

Programming Plans, and specially their key elements, seem to be related to functional information. It

seems clear that in the previous example about the Plan to compute an average, the place where the running total is divided between the number of items is related to what the Plan is meant to do.

Prolog Techniques are concerned with how instantiations of Prolog objects are linked through the program. This characteristic seems to link this model to Data-flow information, while the stress on well known Data Structures and the operations performed over them make Schemas related to Data Structure information.

3 Which structural model?

The experiment described here was concerned with exploring the nature of the mental model Prolog programmers build. It investigated this by finding out which structural model is most relevant for them. The comparisons were made taking into account the key elements of these structural models.

To measure the relevance of a specific structural model, the experiment considered a recall task similar to the one in Wiedenbeck (1986). Subjects were asked to understand and memorise a small Prolog program, and then recall what they could of it. This code was analysed in terms of the different models of structural knowledge of Prolog and their associated key segments. The relative success of recollection of the different key segments was compared against the relative success of recollection of the rest of the program to establish the relevance of these structural knowledge models for Prolog programmers. The main difference with Wiedenbeck's study is that the present experiment compared several structural models for program comprehension, while Wiedenbeck's only took into account Programming Plans. The structural models taken into account in this experiment are Plans, Prolog Techniques, Prolog Schemas and Recursion Points. This last model highlights Control-flow information, and is concerned with how recursion is handled in Prolog.

Additionally, and to confirm differences due to experience, there was a function identification task in the experiment. Besides recalling the code, the programmer subjects were asked to describe the program's function. The accuracy of these descriptions was compared for novice and experienced subjects to confirm whether there were any differences between these two groups in terms of their program comprehension skills.

Note that this experiment was concerned with small programs and with short comprehension sessions. In professional, and some times even in academic environments program comprehension is a task that is normally performed over large programs and in extended periods of time. The average length of the experimental programs was 24 lines and the amount of time that subjects were allowed to study them was 3 minutes.

3.1 Aims

The aim of this experiment was to find out for which model of structural knowledge there is a difference in accuracy of recall between key and non-key segments. This finding might suggest which structural model seems to be more relevant to Prolog programmers and therefore will provide information about the nature and characteristics of Prolog mental models.

3.2 Design

This experiment considered one independent variable, level of programming skill (experienced programmers, novice and non-programmer) and nine dependent variables, the success of recollection for the key segments and the non-key segments of four different structure models of comprehension (Plans,

Prolog Techniques, Prolog Schemas and Recursion Points) and the accuracy of Function identification by the programmer subjects.

3.3 Subjects, procedure and materials

There were 30 subjects: 10 experienced programmers and 10 novice Prolog programmers and a group of 10 non-programmers. The group of experienced programmers had at least three years of Prolog experience and were either university lecturers or research fellows. The group of novices had taken a three month introductory course in Prolog and were either undergraduates or masters students. The group of non-programmers did not know anything about computer programming. The novice population was inexperienced in Prolog, but not in programming in general. Most of them knew three or more programming languages apart from Prolog. Also, they often had more recent contact with Prolog than some of the experienced programmers.

This experiment used a control group, the group of non-programmers, because recollection experiments might confound pure memorisation and real comprehension of the code.

The novice and experienced programmer subjects of this experiment performed three similar sessions. In each session, they were given a hardcopy of the experimental program and were asked to study and memorise it. This study period lasted 3 minutes. After this, the subjects were given 5 minutes to recall and write down what they could remember of the program. Finally, these subjects used another period of 3 minutes to write down a short explanation of what, according to them, the program did.

The control group of non-programmers followed a slightly different procedure. They were not instructed to comprehend but only to memorise the programs. Also, they were not asked to write down an explanation of what the programs did.

In each case the order of presentation of the experimental programs was randomised.

There were three experimental programs. These were, a Prolog version of the 'rainfall' program (Davies, 1994), of the bubble sort and a program that performs a binary to decimal conversion. Figure 5 shows the Prolog version of the 'rainfall' program.

These programs were analysed in terms of key segments of Plans, of Prolog Schemas and of Prolog Techniques according to the definitions by Rist (1995), Gegg-Harrison (1991) and Bowles and Brna (1993) respectively. In the case of Prolog Schemas, the key segments were considered as the compulsory elements of Gegg-Harrison's definition. The choice of key segments for the case of Prolog Techniques was not obvious, so the experiment considered the whole occurrences of the instances of Techniques. The programs were also analysed in terms of their Recursion Points, and the key segments this time were considered as the lines where recursion was invoked or where it stopped. Figure 5 shows the occurrences of the key segments for the case of Plans and Prolog Schemas in the 'rainfall' program.

Finally, as the experimental task included the identification of the programs' functionality, 'disguised' versions of these programs were presented to the subjects. The criteria to 'disguise' these programs is similar to the one used by Wiedenbeck (1986).

3.4 Results

The data of this experiment was analysed in two parts. First, the performance of the programmer groups was compared in terms of the accuracy of the identification of the programs' functions. In the main part of the experimental analysis, the percentage of recollection of key and non-key segments was compared for each one of the four structural models considered.

```
/* average(-,-) */

average(Average,Max):-
        read_rain(RainList),
        total_rain(RainList,0,Sum,0,TotalDays,0,Max),
        Average is Sum / TotalDays.

read_rain(RainList):-
        write('enter rainfall'),
        read(Rain),
        next_value(Rain,RainList).

next_value(99999,[]).

next_value(Rain,[Rain|Rest]):-
        Rain =\= 99999,
        write('enter rainfall'),
        read(NewRain),
        next_value(NewRain,Rest).

total_rain([],Sum,Sum,TotalDays,TotalDays,Max,Max).

total_rain([Rain|Rest],InSum,OutSum,InTotalDays,OutTotalDays,InMax,OutMax):-
        max(InMax,Rain,TempMax),
        TempTotalDays is InTotalDays + 1,
        TempSum is InSum + Rain,
        total_rain(Rest,TempSum,OutSum,TempTotalDays,OutTotalDays,TempMax,OutMax).

max(Max,Min,Max):-
        Max >= Min.

max(Min,Max,Max):-
        Min < Max.
```

Figure 5: Key segment occurrences for Schemas (**in bold**) and for Plans (*in italic*) for a version of the 'rainfall' program.

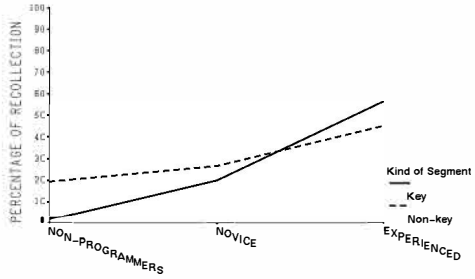

Figure 6: Percentage of recollection for key segments of Schemas and lines outside them

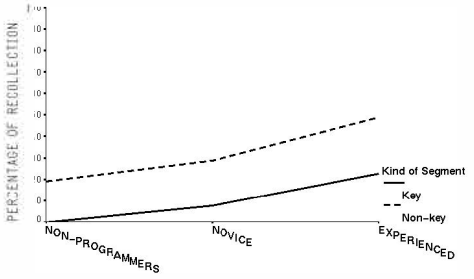

Figure 7: Percentage of recollection for key segments of Plans and lines outside them

In the first case, the programmers' statements were considered as correct only if they mentioned the major functions performed by the programs. It was not surprising that the experienced programmer group was more successful in identifying the function of the programs. Their percentage of correct identification was 70%, while for the novices it was 23.3%. A Chi-square test showed that this difference was significant ($F(1) = 15.15$, $p << .01$).

As mentioned earlier, this first part of the analysis was performed to confirm that there were differences in the degree of understanding of experienced and novice programmers.

The hand written record of the subject's recollection of the code was the raw data for the main part of the experimental analysis. This analysis compared the percentages of recollection of the occurrences of the different kinds of key segments versus the percentages of recollection of the program lines that did not contain occurrences of these key segments. For example, it can be seen in figure 5 that the lines 1 to 8, 11, 12, 13, 17, 18, 19 and 21 to 24 do not share elements with the instances of Schemas. Therefore the analysis for Schemas compared the percentage of recollection of these lines (and similar lines in the other programs) with the percentage of recollection of key segments of Schemas. Figures 6, 7, 8 and 9 illustrate the results of these comparisons for the four kinds of structures.

The statistical analysis for this part of the study focused on the rate of change across the subject groups of the difference between the recollection of key segments of structures and lines outside them for each one of the considered structures. For example, it can be seen that for the case of Schemas (figure 6) this difference is negative for non-programmers and for novices, and positive for experienced programmers. It is therefore likely that the rate of change of this difference (interaction effect) is significant. So the statistical analysis for each structure considered one independent variable (level of skill), and one dependent variable (Key-Non-key segment percentage of recollection difference). For each one of the four comparisons, a one-way ANOVA analysis was run after verifying that its assumptions had been met. The only case for which this rate of change among groups was significant was for Schemas ($F(2,29) = 8.57$, $p < .05$).

This analysis only considered the interaction effects because, as it can be seen in figures 6, 7, 8 and 9, Non-key segments were in general easier to remember than Key segments. This effect had to do with the fact that the different kinds of key segments had different average sizes, and some of them were considerably larger than the average line of code. While some kinds of key segments, for example, typically had short base cases as instances, some others had instances that comprised several long lines. Another factor that contributed to this disparity in recollection was that the location of some key segments in the code was not balanced (some of them tended to appear at the bottom of the program).

Although a direct comparison across key segments of the different structures showed that again Schemas was the most relevant structure for programmers, this comparison was not considered as statistically

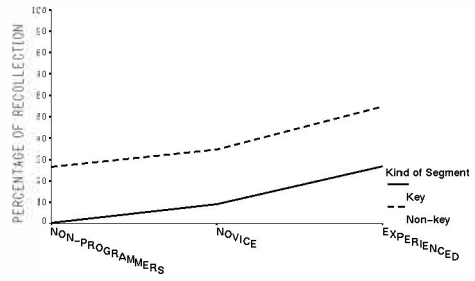

Figure 8: Percentage of recollection for key segments of Techniques and lines outside them

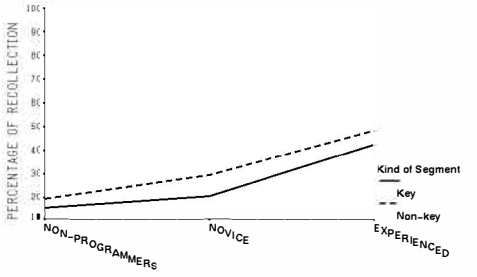

Figure 9: Percentage of recollection for key segments of Recursion Points and lines outside them

reliable because some of these key segments were easier to remember than others. The causes of this effect are basically the same that made Non-key segments easier to remember than Key segments.

3.5 Discussion

The results show that Schemas, stereotypical patterns of programming procedures related to Data Structure aspects, seem to be important for Prolog program comprehension. When comparing the difference between the percentage of recollection of key and non-key segments for each structure, Schemas were the only case for which the difference between groups was significant. It seems that these results show that Schemas become more important for the comprehension process as Prolog programmers develop higher levels of skill.

Following Brooks's hypothesis, it could be said that Schemas seem to be the key elements of structural knowledge that Prolog programmers use to guide their comprehension process. This contrasts with procedural languages, where Plans and their key elements seem to be important (Wiedenbeck, 1986; Davies, 1994).

The finding that information related to Data Structures is important for the comprehension process is in agreement with the results of Bergantz and Hassell (1991). They found that 'data structure relationships play a dominant role at the beginning of the comprehension process' (p. 323) for the case of Prolog. Although the period they considered as the beginning of the comprehension process was approximately three times of what the present experiment took into account (ten minutes as opposed to three minutes) and the experimental task was different (program modification), the basic finding is quite similar.

It is interesting to compare these results with those obtained by Wiedenbeck (1986) because the experiment reported in this paper is similar to hers. She found that key segments of Plans were relevant for the case of Pascal. Figure 7 can be used to make a closer comparison. With a graph similar to this one, Wiedenbeck shows how this type of functional information is very important only for experienced programmers. Her results were not replicated in the present study. It could be argued that the experimental programs were different, but the results when considering only the bubble sort program, which is similar to the sort program Wiedenbeck uses, are basically the same to those obtained when taking into account all three programs. So it seems that the key difference is the programming language considered, although taking a closer look at this language's properties and at the experiment's characteristics might offer a more precise explanation for this difference in the results.

It seems reasonable to think that in absence of any other information (neither internal nor external documentation, and with cryptic variable and procedure names) patterns of typical operations

performed over familiar data structures can be very important to start making sense of the code. This lack of documentation and meaningful variable names seems to be an important issue for Prolog. Green, Bellamy, and Parker (1987) mention that Prolog, due to its poor 'role-expressiveness', is specially sensitive to naming style ('Salient variable names are almost the only method of making a Prolog program "role-expressive" and thereby revealing the plan structures', p. 142). An obvious question is how naming style influences the program comprehension mental model, or in other words, which aspect of the program (Data-flow, Control-flow, Data Structure or Function) would be relevant for programmers when meaningful variable names are considered.

4 Conclusions

This paper reports an experiment that explored which one of four programming knowledge structures seems to be important for Prolog programmers at the early stages of the program comprehension process. These structures are related to Functional, Data-flow, Data Structure and Control-flow information. It seems that information related to data structures is important in this case. The experiment involved a program comprehension and memorisation task followed by the recollection of the program by three groups of subjects: experienced programmers, novices and non-programmers. There were significant differences when comparing the performance of these three groups only for the case of Schemas, a structure that emphasises Data Structure relationships. These results suggest that Data Structure relations are important for the initial comprehension process of Prolog programmers.

The results of this experiment suggest that the mental model that Prolog programmers build when doing program comprehension is different from the one that programmers of procedural languages construct. The former seems to be influenced by Data Structure relations while the latter, according to Wiedenbeck (1986) and Davies (1994), is related to Functional information. This conclusion needs to be confirmed and its importance needs to be related to a more common programming task such as debugging or program modification.

Acknowledgments

The author would like to express his thanks to Ben du Boulay of COGS, Sussex University and to Thomas Green of CBL, Leeds University for their help in the preparation of this paper. Thanks also to all the subjects for their time and patience.

References

Bellamy, R. K. E., & Gilmore, D. J. (1990). Programming plans: Internal and external structures. In Gilhooly, K., Keane, M. T. G., Logie, R. H., & Erdos, G. (Eds.), *Lines of thinking: Reflections on the psychology of thought, Vol 1.* Wiley, London, U.K.

Bergantz, D., & Hassell, J. (1991). Information relationships in PROLOG programs: how do programmers comprehend functionality? *International Journal of Man-Machine Studies, 35*, 313–328.

Bowles, A., & Brna, P. (1993). Programming plans and programming techniques. In Brna, P., Ohlsson, S., & Pain, H. (Eds.), *World conference on artificial intelligence in education*, pp. 378–385 Edinburgh, UK. Association for the advancement of computing in education.

Brna, P., Bundy, A., Todd, T., Eisenstadt, M., Looi, C. K., & Pain, H. (1991). Prolog programming techniques. *Instructional Science, 20*(2), 111–133.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies, 18*, 543–554.

Davies, S. P. (1990). The nature and development of programming plans. *International Journal of Man-Machine Studies, 32*, 461–481.

Davies, S. P. (1994). Knowledge restructuring and the acquisition of programming expertise. *International Journal of Human Computer Studies, 40*, 703–726.

Detienne, F. (1990). Expert programming knowledge: a schema-based approach. In Hoc, J., Green, T. R. G., Samurçay, R., & Gilmore, D. J. (Eds.), *Psychology of Programming*. Academic Press, Ltd., London, U.K.

Gegg-Harrison, T. S. (1991). Learning Prolog in a schema-based environment. *Instructional Science, 20*, 173–192.

Gilmore, D. J., & Green, T. R. G. (1988). Programming plans and programming expertise. *Quarterly Journal of Experimental Psychology, 40A*, 423–442.

Green, T. R. G., Bellamy, R. K. E., & Parker, J. M. (1987). Parsing and Gnisrap: a model of device use. In Olson, G. M., Sheppard, S., & Soloway, E. (Eds.), *Empirical Studies of programmers, second workshop*, pp. 132–146 Norwood,NJ. Ablex.

Ormerod, T. C., & Ball, L. J. (1993). Does design strategy or programming knowledge determine shift of focus in expert Prolog programming? In *Empirical Studies of programmers, fifth workshop*, pp. 162–186 Norwood,NJ. Ablex.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology, 19*, 295–341.

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science, 13*, 389–414.

Rist, R. S. (1995). Program structure and design. *Cognitive Science, 19*, 507–562.

Wiedenbeck, S. (1986). Processes in computer program comprehension. In Soloway, E., & Iyengar, S. (Eds.), *Empirical Studies of programmers, first workshop*, pp. 48–57 Norwood,NJ. Ablex.