Focal Structures and programming information types

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 related to functional information known as Programming Plans. It also has been claimed that experienced programmers organise this representation in a hierarchical structure where some elements of plans are focal or central to them. However, it is not clear that this is the case for other types of programming language, especially for those which are significantly different from the procedural paradigm.

The study reported in this paper investigates whether these claims are true for Prolog, a language which has important differences to procedural languages. Prolog 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 how programmers comprehend Prolog code in a significant way.

The findings of the study suggest that Plans and functional information are important for Prolog programmers, but that there is also at least another model of structural knowledge valid for this language. This model of structural knowledge, Prolog Schemas, is related to data structure information and it seems that a hierarchical organisation that highlights the relevance of some elements as focal is still valid for Prolog. These results support the view that comprehension involves the detection of varying aspects of the code and that each of the structures related to these aspects might have their own organisation and hierarchical relations.

Keywords: program comprehension, focal lines, Prolog.

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.

Programmers are said to be able to comprehend computer programs because of the structural and strategic knowledge about programming that they have acquired through experience. The Programming Plan concept has been the dominant model used to represent the structural knowledge that programmers possess (Detienne, 1990). Programming plans are said to be strongly related to what the program does (functional information). It has been suggested that this model has a close relationship to the mental models programmers build when they perform program comprehension. Also, it has been claimed that plans typically comprise several elements, and that through experience, programmers organise this structural knowledge in such a way that some elements become focal or salient (Davies, 1993, 1994).

*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)

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, and therefore suitable to explore these claims is Prolog. This programming language is in a class of its own because of its declarative nature and its powerful primitives. 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 & Gilmore, 1990; Ormerod & Ball, 1993).

This paper explores the nature of Prolog programmers' structural knowledge, adopting the model proposed by Davies (1993, 1994), but extending the notion of focal structures to consider not only function but also other programming information types. This investigation is performed through two experiments, the first one applies a program recall task to find out which of several structural models are relevant for Prolog programmers, while the second one compares the importance of two specific structural models for this same programming language through a code recognition task.

This document is divided into four parts. The next section gives a brief account of program comprehension studies, the following parts describe and discuss the two experiments and the final section analyses the global results.

2 Program comprehension

Program comprehension is a complex cognitive process that involves the acknowledgement and understanding of several elements. The result of this process is a more or less detailed *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 and the task in hand.

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. According to Pennington (1987) code aspects or Text Abstractions are the different kinds of information that are implicit in the program's text. The programming Plans model, for example, is primarily based on functional and data flow code aspects, this is, in what the program does and the relationships between the program variables.

```
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)

2.1 The comprehension process

One of the earliest theories of program comprehension was introduced by Brooks (1983), who 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 program execution and its relation to the 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 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. The organisation that the application of this model imposes over a particular instance of code is the model *instance*. In the case of Plans, the structural model would be Plan knowledge as an abstract concept and the model instance would be the result of applying a Plan view over a specific program.

One assumption shared by several models is the idea that programmers, especially experienced programmers, are able to organise the program they understand into meaningful chunks. One of such models is Programming Plans (Pennington, 1987; Gilmore & Green, 1988; Davies, 1990) and the meaningful chunks or Plans are 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.

Some studies have suggested that the elements that comprise these units of meaningful information have different degrees of relevance or saliency for programmers (Rist, 1989; Davies, 1993, 1994; Rist, 1995).

The elements that directly encode the goal of a particular Plan are said to be focal for experienced programmers. For example, if the programming task is to accumulate data and compute an average, the key or focal element of it will be the place where the division between the running total and the number of items takes place (Rist, 1995). Although Plans have been related to data-flow as well as to functional information, focal elements of Plans seem to be highly related to function only. Because of the functional nature of the focal elements of Plans, this paper considers Plans as related mainly to function.

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. According to Davies (1993, 1994), as a result of experience, programmers restructure their programming knowledge by identifying the focal elements of plans and assigning them a high place in the plan hierarchy. He gives empirical evidence that supports this notion of focal structures by performing an experiment that compares the recognition of focal and non-focal lines of Pascal programs for three groups of programmers: novices, intermediates and experienced. His results show that experienced programmers, unlike the other groups, are able to recognise more quickly and more accurately focal segments than non-focal segments.

However, studies of Programming Plans and focal structures 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 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 the Prolog programmer's mental model. One of the purposes of this paper is to explore whether such a relationship exists.

Also, there has not been any research about focal structures for models other than Plans. It is interesting to investigate if the restructuring of programming knowledge that Davies refers to only happens for functional information, or whether knowledge related to other types of information is also restructured as a result of programming experience.

```
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)

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. According to Pennington, full comprehension of a program involves the detection of several, complementary aspects of the code.

Programming Plans, and especially 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. The example of Prolog Schema given in Figure 4 shows a typical operation that a procedure performs over a list, one of the most important data structures in Prolog.

The experiments described in this paper employ program recall and recognition tasks to find out which structural model and related aspect of the code is important to Prolog programmers when adopting the knowledge restructuring theory of program comprehension by Davies (1993, 1994), but extending it to consider not only function but also other information types.

3 Which structural model?

The experiment described in this section was concerned with exploring the nature of the mental model Prolog programmers through finding out which structural model of several considered 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 took into account Programming Plans only. 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 the subjects' level of skill, 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 that there were differences between these two groups in terms of their program comprehension skills.

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, 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 and 10 novice Prolog programmers and a group of 10 non-programmers. The group of experienced programmers had on average 8.6 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 the recall experimental task 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

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

average(Average,Max):-
        read_rain(RainList),
        total_rain(RainList,RainAmount,NumberOfDays,Max),
        Average is RainAmount / NumberOfDays.

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

total_rain([],0,0,0).

total_rain([Rain|Rest],RainAmount,NumberOfDays,Max):-
        total_rain(Rest,TempRainAmount,TempNumberOfDays,TempMax),
        RainAmount is TempRainAmount + Rain,
        NumberOfDays is TempNumberOfDays + 1,
        max(TempMax,Rain,Max).

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

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

next_value(99999,[]).

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

Figure 5: A version of the 'rainfall' program highlighting the focal structures of Schemas (**in bold**) and of Plans (*in italic*).

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 was 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.

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. 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 ($F(2,29) = 8.57$, $p < .05$).

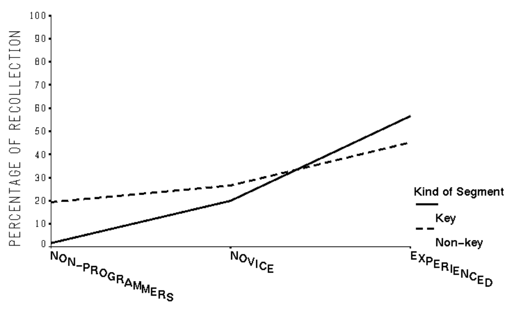

Figure 6: Percentage of recollection for key segments of Schemas and lines outside them

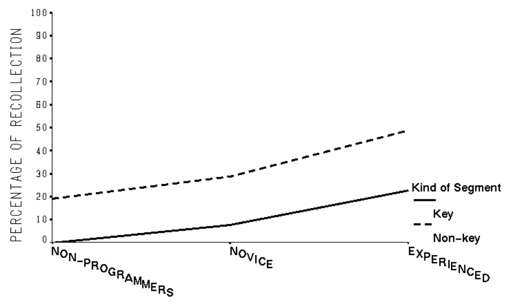

Figure 7: Percentage of recollection for key segments of Plans and lines outside them

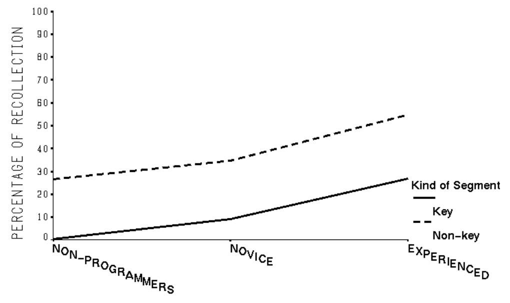

Figure 8: Percentage of recollection for key segments of Techniques and lines outside them

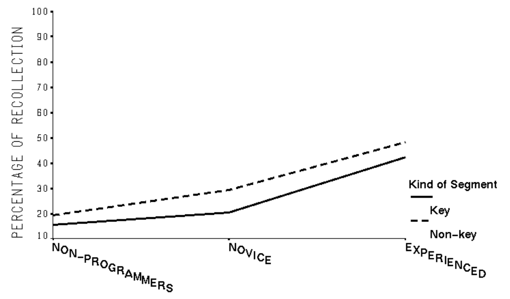

Figure 9: Percentage of recollection for key segments of Recursion Points and lines outside them

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 recalled better 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 those of Schemas were the most relevant for programmers, this comparison was not considered reliable because of the disparity on size and location in the code of the different types of 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 (about data structure) 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 in Prolog 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 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.

Another aspect to consider has to do with a combination of the nature of the recall task and Prolog. Recall was particularly useful for this experiment because several structural models could be explored through the same task, however, there are some characteristics of the recall task that have to be looked at in more detail. According to Kintsch (1970), retrieval is an important subtask in recall and material which is easy to associate and organise is more likely to be retrieved and therefore recalled. Key segments of Prolog Schemas are structures that have well defined internal organisation and that have a high degree of independence from other parts of the program. This is not the case for the key segments of other structures. For example, from Figure 5 it can be seen that the key segments of Schemas can be understood without reference to any variable or procedure name external to them, however, this is not the case for the key segments of Plans. To understand how *RainAmount* is computed, first the variables *TempRainAmount* and *Rain* have to be traced, but these variables are dependent on other segments of the code, among them a part of a Schema key segment. If some of these other segments of code cannot be recalled correctly, it would be much harder to recall this particular Plan key segment. In a way, the Prolog Schemas model slices Prolog programs without 'cutting' any link of Prolog objects associations, creating a chunk that can be called an independent unit and that could therefore be organised, retrieved and recalled more easily. However, this does not mean that Schemas are necessarily relevant to Prolog programmers, or that other aspects of the code are not important to them. These other aspects of the code might be related to segments which are highly associated, and therefore dependent in terms of retrieval, from other parts of the program.

These considerations suggest the need for more experimentation controlling aspects like naming style and modifying the experimental task. The next section describes an experiment that took into account these considerations.

4 A comparison of two structural models

The findings of the program recall experiment suggested that Schemas' key segments, programming constructs related to data structure, seem to be relevant for Prolog programmers, but aspects such as naming style and the experimental task might have played a role in this outcome.

The experiment reported in this section tried to find out whether focal structures (and therefore programming structural knowledge) in Prolog are related to functional or data structure aspects. Functional information was considered because there is strong empirical support, at least from the procedural languages literature, that function is an important aspect of programming knowledge.

The finding of the previous experiment that focal structures of Schemas are important for Prolog program comprehension could be explained by at least three causes (or a combination of them). One possible cause is that the comprehension process normally focuses on functional information, but the use of cryptic variable names and the 'opacity' of the programming language exacerbated the difficulty of detecting functional information. It seems that the most relevant aspect (or perhaps the only one) that can be detected in these circumstances is data structure. This explanation assumes that the aspect of the code which is accessible to programmers determines what segments of the program are considered as focal.

Another possible cause is that the recall, as oposed to some other task, favoured key segments of Schemas and blurred the importance of other code aspects. It is also possible that comprehension in Prolog is linked to data structure, and that therefore data structure information is relevant regardless of factors like naming style and experimental task.

The first explanation would predict that data structure aspects are only important when cryptic naming style is considered. When some other aspects, function among them, are easily available in the code, Prolog programmers would not be different from those of procedural languages and will take functional information as the relevant aspect of the code. The second explanation implies that the choice of a relevant aspect would change when considering another experimental task. In this case, assuming again that Prolog is not that different from procedural languages, a likely outcome is that function is the relevant aspect regardless of experimental conditions. The last explanation would predict that as data structure is the relevant aspect in Prolog, this information type would be the most important for Prolog programmers regardless of experimental task and conditions.

The experiment described in this section tried to find out which one of these explanations is more likely for the case of Prolog by using a recognition experimental task.

4.1 Hypotheses

The choice of focal structures in program comprehension is affected by several factors, programming language and naming style among them. Therefore, it is a combination of these factors that determines what programmers consider as focal structures in a specific case. Focal structures related to data structure aspects will be relevant for poor naming style experimental conditions. However, there will be no structures that can be considered as focal for the case of meaningful naming style.

4.2 Design

This experiment was a 2 X 2 X 4 factor design with three independent variables (expertise level, naming style and kind of structure and two dependent variables (response time and success level of recognition). The expertise levels were expert and novice, the naming styles were meaningful and cryptic and the kinds of structures were focal according to functional aspects, non-focal according to functional aspects, focal according to data structure aspects and non-focal according to data structure aspects.

4.3 Subjects and procedure

The subjects of the experiment were 20 novices and 20 experienced Prolog programmers. The novice population comprised undergraduate students who had taken a one term introductory course in Prolog. Most of them were second or third year students and knew at least two more languages apart from Prolog. As in the first experiment, the novice population was inexperienced in Prolog but not in programming in general.

The experienced programmer population had on average 11 years of Prolog programming experience and had written programs longer than three thousand lines (on average). They were either lecturers or researchers at academic institutions and eight of them had taught a Prolog course.

As the experimental materials of this experiment were similar to the ones of the experiment reported in the previous section, the subjects in these two experiments were different.

The experimental procedure was very similar to the one applied in Davies (1994). Each programmer performed four code recognition sessions. In each of these sessions, they were presented with a short Prolog program for a period of three minutes. They were asked to study and memorise this program. Following this presentation, sixteen program segments were displayed on the screen and the subjects had to indicate whether the segment was in the program they had studied or not. Their answer and response time was recorded. These four sessions collected data about meaningful and cryptic coding styles, focal structures of plans and Prolog schemas, and in each of these structures about focal and non-focal segments of code. Of the four sessions, two were for programs with poor naming style and two for programs with meaningful naming style. In each session, eight of the program segments that the subjects had to recognise were related to focal structures of plans and the other eight to Prolog schemas. For each of these eight segments, four belonged to the program they studied and the other four to programs that performed similar tasks. Also, only four of these eight segments were focal structures, either from the program they studied or from the similar versions.

Non-focal structures, either from plans or from schemas, meant any segments of code which were not focal for its own kind. This means that non-focal structures of one kind could include focal structures of the other kind. This was because there could not be a direct comparison between focal structures of plans and those of schemas. Therefore, focal structures of plans (and of schemas) had to be compared against everything outside them. They could not be compared directly because focal structures of plans involve a small number of elements and comprise only one line, while focal structures of schemas are scattered through several lines and clauses and include more elements.

Additionally, after each recognition session subjects were presented with eight questions relating to functional and data structure aspects. There were two reasons to include this subtask in the experimental session. First, to verify that there were differences in the quality of comprehension between the novice and experienced groups and also because there was an implicit assumption in the formulation of the hypothesis about the type of information that naming style, among other factors, hides or makes accessible. It is assumed that one of the effects of a cryptic naming style, especially for a language which is very sensitive to this aspect, is that information about function is obscured while information about data-structure is made accessible. If this is in fact the case, the subjects, especially the experienced ones,

```
1)
Average is RainAmount / NumberOfDays.
2)
next_value(NewRain,Rain,Rest).
3)
 total_rain([],0,0,0).

 total_rain([Rain|Rest],RainAmount,NumberOfDays,Max):-
          total_rain(Rest ,TempRainAmount,TempNumberOfDays,TempMax),

4)
 total_rain([Rain],Rain,0,0,0).

 total_rain([Rain|Rest],TempRain,RainAmount,NumberOfDays,WetDays):-
          total_rain(Rest,TempRain,TempRainAmount,TempNumberOfDays,TempWetDays),
```

Figure 10: Probe segments of code for the 'rainfall' program.

would be able to answer questions about data-structure more accurately than those related to function only for the case of programs with cryptic naming style.

4.4 Materials

This subsection discusses the programs, the code segments and the questions that were used in the experiment.

There were four experimental programs. The first one of these transforms sublists of digits into decimal numbers and then sums them up. The second is a version of the bubble sort, the third implements a decimal to binary conversion and the fourth is a Prolog version of the 'rainfall' program. All these programs are around 23 lines in length, perform a specific calculation on their own and have well defined input and output values. As one of the interests in the experiment is to compare comprehension of programs with different naming styles, each one of these programs had two versions, one with meaningful procedure and variable names and the other with cryptic naming style for these elements. Three of these programs were basically the same as those which were used for the experiment reported in the previous section. The program shown in Figure 5 was also used for this experiment.

The four experimental programs were analysed in terms of their focal segments of Plans according to the definition by Rist (1995) and in terms of their focal segments of Schemas according to the definition by Gegg-Harrison (1991), taking as the focal segments those elements that Gegg-Harrison considers as compulsory. As can be seen in the program in Figure 5, and as is indeed the case for the other experimental programs, the focal segments of Plans and Schemas do not have elements in common. It can also be seen that while focal segments of Plans normally comprise a single line, focal segments of Schemas are usually scattered through several lines of code.

After studying the code for three minutes, subjects were presented with sixteen probe segments to decide whether they belonged to the program they just had seen or not. Some of the code segments for

1. Is one of the tasks of the program to obtain the number of input data items supplied by the user?
2. Does the program find out which is the input number with the maximum number of occurrences?
3. Does the main processing of the elements of the list in the total_rain/4 procedure happen in a back to front order?
4. Is the list in the total_rain/4 procedure processed until a certain element is found?

Figure 11: Some comprehension questions for the 'rainfall' program.

the 'rainfall' program are shown in Figure 10. The segments that did not belong to the displayed programs belonged to programs that performed similar tasks.

To ensure that the difficulty of the recognition task was similar for all the probe items, and therefore that any differences in the subjects' performance were due to the type of probe items rather than to other factors, the probe items were controlled for several aspects. These controls ensured that the probe items, when divided into subgroups according to the experimental conditions (focal or non-focal, schema or plans and belonging or not belonging to the displayed program) had a similar recognition difficulty. The criteria to establish this similarity was based on the serial position of the code segments in the program, on their length and, for the case of those that did not belong to the displayed program, on the degree to which they resembled a code segment of this program. The probe items were evenly distributed with respect to their serial position in the code, they had roughly the same size and the distractor items had approximately the same same degree of similarity towards the code of the studied program.

4.4.1 Comprehension questions

As mentioned in Section 4.3, the accuracy rate for the comprehension questions would be important in confirming the assumption that naming style influences the kind of information that is available to programmers and therefore the characteristics of their mental model. According to this assumption, when faced with programs with a cryptic naming style, programmers resort to basing their comprehension on data-structure aspects. If this is indeed the case, the accuracy rate for questions related to function should be less than the accuracy rate of those related to data-structure for the case of cryptic naming style. These rates should be similar for the case of programs with meaningful naming style.

There were yes-no comprehension questions for functional and data structure aspects. Some of the comprehension questions for the case of the 'rainfall' program are presented in Figure 11. The first two questions refer to functional aspects while the last two are related to data structure issues.

4.5 Results

The results of the experiment comprise analyses for the case of accuracy and response time of both segment recognition and answering of the comprehension questions. For each one of these four cases repeated measures ANOVAs for skill level as between subjects condition and naming style, kind of structure (Plans or Schemas) and segment category (focal or non-focal) as within subjects variables were run after verifying that their assumptions had been met.

The results of the segment recognition accuracy are summarised in Figure 12. It can be seen that experienced Prolog users, unlike novices, had a better accuracy of recognition for focal segments than for non-focal segments. In general, experienced users were better than novices in the recognition task,

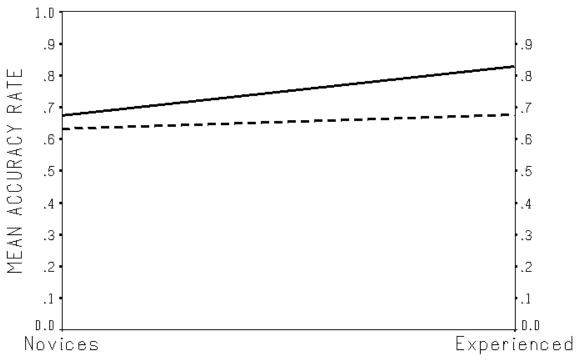

Figure 12: Probe item recognition accuracy rate by novices and experts.
(——) focal segments; (- - -) non-focal segments.

F(1,38) = 16.94, p < .01; and key segments were better recognised than non-key segments, F(1,38) = 44.93, p < .01; but there were interaction effects for these two conditions, F(1,38) = 13.81, p < .01. There was an absence of interaction effects between segment categorisation, naming style and kind of structure (Schemas or Plans), which means that experts were more accurate in recognising focal rather than non-focal segments regardless of naming style and of the kind of information type (function-Plan, data structure-Schema).

The results for the segment recognition response time show main effects for skill level, F(1,38) = 4.24, p < .05; and for segment category, F(1,38) = 7.85, p < .01, indicating that in general experts were slower than novices in discriminating code segments and that key segments were recognised more quickly than non-key segments. Interaction effects near to significant for these two conditions showed a tendency for the difference in key non-key segment recognition to be larger for experienced users than for novices, F(1,38) = 2.92, p = .096 (Figure 13).

The results of the accuracy of comprehension question answering show that experts had better average accuracy rate, F(1,38) = 56.8, p < .01 and that the meaningful naming style was more helpful to all programmers, F(1,38) = 7.75, p < .01.

The results of the response time for the question answering experimental task show that questions about function were answered quicker than questions about data structure for all the experimental conditions, F(1,38) = 5.58, p < .05.

4.6 Discussion

In general, the results of the experiment do not support the view that naming style is important in determining the kind of focal structure and therefore information type preferred by Prolog programmers. Both kinds of information, function and data structure, seem to be important for experienced Prolog programmers regardless of naming style. Experienced Prolog programmers, unlike novices, seem to make a distinction between key and non-key structures, marking the former as more relevant. This distinction is not affected by the kind of key segment considered (Schemas or Plans) or by the naming style of the programs they comprehend.

This result is somehow surprising because the hypothesis contained the implicit assumption that there could only be one kind of focal structure when comprehending a program. The experimental results seem to indicate that programmers might consider different kinds of segments as focal depending of the

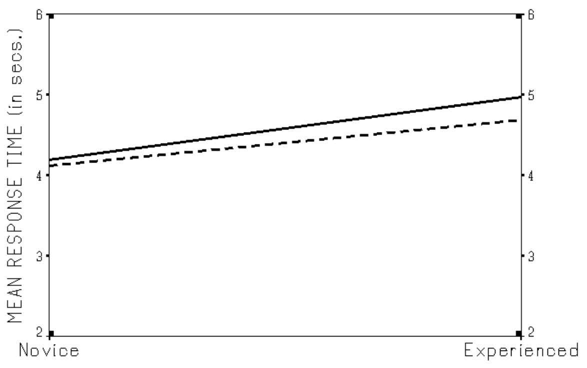

Figure 13: Probe item recognition response time by novices and experts.
(——) focal segments; (- - -) non-focal segments.

aspect of the code they are focusing on. If they are trying to understand data structure issues, a set of program segments would be relevant for them, but if they are trying to decode functional information, they might consider a different set of code segments as important.

At least for this experiment, key segments of Plans and of Schemas seem to be relevant for Prolog programmers. This seems to indicate that function as well as data structure aspects are important for this programming language. However, there might be other aspects also relevant for Prolog programmers. It might also be the case that also aspects such as control-flow or data-flow are important and have their own focal structures.

The experiment reported in this section replicated Davies (1994) results globally, but there are several differences. First, Davies only considered focal structures of Plans, and the present experiment shows that focal structures might not be restricted to Plans and therefore to functional aspects, and even more importantly, that several kinds of structures might be considered as relevant by programmers. Another difference is that in this experiment, experienced programmers were in general slower than novices in the segment recognition task, while in Davies' study the former group were quicker than the latter. It could be said that as experienced programmers had a higher accuracy rate than novices, the response time difference seems to be a speed/accuracy trade off. However, this does not explain the discrepancy of results with Davies' study.

It has to be said that the experimental settings in these two experiments were not identical and some of these differences might account for the different results in response time. One of the most important differences is that Davies' subjects had a longer period of time to study the program (ten minutes as opposed to three minutes). This difference in study time was chosen because in pilot sessions, experienced programmers, and even some novices, found ten minutes to be too long a time to comprehend the programs. Three minutes seemed to be just enough for them. However, it is difficult to understand how a longer exposure to the experimental programs might had made a difference in making experienced users quicker than novices. Apart from this difference, the results support the finding that focal segments are quicker to recognise and there is a tendency for experts to show a more pronounced difference in this recognition time.

The question answering task was important to establish that there was a difference in comprehension performance between experienced and novice Prolog users. Also, the lack of interaction effects for kind of structure and naming style seem to be in agreement with the similar results (no interaction effects) for segment categorisation, kind of structure and naming style for the segment recognition task. The experimental hypothesis predicted that naming style would have an influence over the kind of information that could be detected in the program, and that therefore both the recognition and question answering task would have different results for different naming styles and information types. Just as the difference between focal and non-focal segments remained constant for both of these conditions, the question answering accuracy did not vary depending on both naming style and kind of structure. There were main effects for naming style, which indicate that cryptic programs are more difficult to understand than meaningful code, but it seems valid to assume that in both cases programmers find both kinds of key segments relevant.

5 General discussion

Both experiments support the notion of knowledge restructuring proposed by Davies (1993, 1994) that states that experienced programmers restructure their structural knowledge to allow some elements of it to become relevant or salient. However, the results of these experiments indicate that this knowledge restructuring is not exclusive to Plans and functional information. For the case of Prolog, programmers also seem to restructure their programming knowledge according to at least another type of programming information (data structure). It seems also that this restructuring of several aspects of programming knowledge might produce several kinds of key segments in the code.

The experiments reported here show that information about data structure and Prolog Schemas, an associated model of this type of structural knowledge, are important for Prolog programmers and that they restructure this kind of programming knowledge as a result of experience.

The finding that there might be more than one kind of focal structure in a program might seem counter intuitive at first. It would seem that only one aspect could be relevant to programmers. However, the results of these experiments seem to support the idea that comprehension involves the detection of several aspects of the program, and each of these aspects might have its own organisation and hierarchical relations. Which aspects are relevant and to what degree might depend on the programming language and the programmer's experience, among other factors. For Prolog programmers, it seems that function and data structure are two important aspects, although there might be more than these two relevant information types.

One issue that remains to be explained is the reason for the discrepancy in the experimental results concerning Plan focal structures. In the first experiment, the only relevant key segments were those of Schemas, while in the second experiment both key segments of Schemas and of Plans seemed to be relevant. Both experiments had similar settings and the programmer subjects had similar experience and background. One of the differences between these two experiments was the nature of the experimental task. The first experiment used a program recall task while the second applied a program recognition task. As mentioned in Section 3.5, according to Kintsch (1970), one important difference between these two memory tasks is that recall, unlike recognition, is sensitive to material that can be associated and organised clearly.

Key segments of Plans, unlike those of Schemas, do not seem to have a well defined internal organisation because they are dependent from other parts of the program. If these other parts of the program could not be recalled successfully, it would be harder for key segments of Plans to be recalled accurately. In the second experiment, on the other hand, Plan key segments could be recognised even when other parts of the program necessary for their complete understanding were not present. So the disparity of results could be explained by a combination of the experimental task and the ***hidden dependencies*** (in terms of Green's(1999) cognitive dimensions) of Prolog.

Further support for the notion of this sensitivity of results to the experimental task is provided by Navarro-Prieto and Cañas (1998), who report differences even in the results of recognising to accept (when the probe segment belongs to the studied program) and recognising to reject (distractor items).

The global results of the study indicate that Schemas are a plausible model of structural knowledge for Prolog, but functional information, and Plans, are important as well for Prolog programmers. This result does not rule out the possibility that there might be further aspects of the code that are important for this programming language.

6 Conclusions

This paper reports a study that explored which programming aspect seems to be relevant for Prolog programmers adopting the model of structural knowledge organisation proposed by Davies (1993, 1994), but extending the notion of focal structures to consider not only function but also other programming information types.

The study reported in this paper comprised two experiments. The first one was of an exploratory nature and tried to find out which one of four programming knowledge structures seems to be important for Prolog programmers when doing program comprehension. It seems that information related to data structures was important in this case. This first 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 suggested that data structure relations are important for the comprehension process of Prolog programmers.

The second experiment tried to replicate the results of the first one this time by applying a recognition instead of a recall task taking into account only two kinds of information types, Schemas and Plans. This experiment was intended to clarify the role of naming style in Prolog program comprehension. For this experiment, two groups of Prolog programmers, novices and experienced, tried to understand and memorise several Prolog programs. After this study time, they tried to discriminate program segments as belonging to the studied programs or not. There were significant differences for these two groups when comparing their recognition accuracy. Experienced programmers were able to recognise segments categorised as focal better than novices, regardless of kind of structure (Plans or Schemas) or of naming style. The results of this experiment indicate that at least for the case of Prolog, experienced programmers restructure the organisation of their Plan and Schema knowledge to allow that certain elements of these structures become more relevant than others.

The reason for plan information not being apparently important in the first experiment seems to be due to the difference in the two experimental tasks used in this study, recall and recognition. It has been argued that recall facilitated high rates of accuracy for Schemas but not for Plans. Therefore, it is possible that some other structural models and information types besides Plans and Schemas might be important for Prolog programmers.

Experienced programmers seem to construct several views of the program, according to the different types of information implicit in the program text. It is likely that their structural knowledge for each of these aspects of the code shows an organisation in which specific elements are more relevant than others. This would imply that the notion of focal structures involves not only function but other types of programming information.

Acknowledgments

The author would like to express his thanks to Ben du Boulay of COGS, Sussex University for his help in the preparation of this paper, to Thomas Green of CBL, Leeds University, for his advice in the design of the first experiment and to Judith Good of HCRC, Edinburgh University, and Chris Taylor, of COGS, Sussex University, for their help in contacting Prolog experienced users. 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. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies*, *39*, 237–267.

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. (1999). Building and manipulating complex information structures: issues in prolog programming. In Brna, P., du Boulay, B., & Pain, H. (Eds.), *Learning to build and comprehend complex information structures: Prolog as a case study.* (in press).

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.

Kintsch, W. (1970). *Learning, memory and comceptual processes.* John Wiley & sons, inc.

Navarro-Prieto, R., & Cañas, J. J. (1998). Mental representation and imagery in program comprehension. In Green, T. R., Bannon, L., Warren, C., & Buckley, J. (Eds.), *Proceedings of the Ninth European Conference on Cognitive Ergonomics* Limerick, Ireland.

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.