The Role of Notation and Knowledge Representation in the Determination of Programming Strategy: A Framework for Integrating Models of Programming Behavior

SIMON P. DAVIES
Huddersfield Polytechnic, United Kingdom

A number of accounts of expert programming behavior have been advanced. These models of the programming activity have served to highlight the range of factors that are thought to underpin programming strategy. However, such accounts have tended to emphasize either the effects of the organization of the programmer's knowledge representation or the role played by features of the notation of the task language on the emergence, development, and support of particular forms of strategy. Such work has neglected to (a) provide an account of the way in which these factors might interact to determine programming strategy, and (b) shed light upon the nature of the development of particular strategies as programming skill increases.

This article presents the results of an empirical analysis of the strategies employed by programmers of varying skill levels using different programming languages. The development of particular forms of strategy is hypothesized to be related both to the development of systematic asymmetries in programmers' generic plan-based representations of programming knowledge and to the way in which features of the notation of particular programming languages might differentially support particular strategies. However, the results of the study reported here suggest that these effects must be interpreted within a broad developmental context. Hence, as programming skill develops, it appears that the notation of the task language tends to take precedence as a determinant of strategy, but has less relevance at beginning stages of skill development and within the context of expert performance. This finding would not be predicted by current models of the programming activity because such models suggest that any notational effects will tend to be consistent regardless of the skill level attained by the programmer. This article endeavors to extend current models of

I would like to thank John Findlay, University of Durham, Adrian Castell, University of Salford, and two anonymous referees for their helpful comments on an earlier version of this article.

Correspondence and requests for reprints should be sent to Simon P. Davies, Department of Psychology, University of Nottingham, University Park, Nottingham, NG7 2RD, UK.
programming by emphasizing the need to consider in detail not only the development of programming strategy, but also the way in which knowledge representation and features of the task language interact to give rise to particular forms of programming behavior.

1. INTRODUCTION

1.1 The Plan-Based Approach to Program Understanding

The idea that experts develop "chunks" of knowledge that represent important functional units or structures within a particular domain has common psychological currency (Chase & Simon, 1973; de Groot, 1965). Within the programming domain, a range of studies has suggested that programming expertise might be characterized by the programmer's ability to decompose and represent programs in terms of chunks of knowledge based upon semantically meaningful elements of programs. For example, Shneiderman (1976) showed that the standard results obtained in the now well known chess studies (de Groot, 1965) are replicable in the context of programming. In addition, a number of other studies, employing more complex recall procedures, have provided additional support for the chunking hypothesis (Adelson, 1981; Barfield, 1986; Bateson, Alexander, & Murphy, 1987; McKeithen, Reitman, Rueter, & Hirtle, 1981).

A number of authors have taken these ideas further and have addressed questions relating to the content of expert knowledge, in addition to its organization. For example, Rist (1986, 1989) and Spohrer, Soloway, and Pope (1985) suggested that the "programming plan" should be regarded as the major characteristic of programming expertise. Such plans are thought to represent aspects of the programmer's knowledge of generic and stereotypic fragments of programs that correspond to specific task goals or subgoals. For example, such plans might include a "running total loop plan" (essentially a control-flow plan that keeps track of a running total), or a "data guard plan" (a plan that checks for invalid input). Though these plans are conceptually separate, they may share a number of common program statements. It is suggested that experts have not only developed a greater range of these plans than novices, but also that experts know the rules which govern the valid application of plans in particular circumstances: so-called "rules of programming discourse" (Leventhal, 1987; Soloway & Ehrlich, 1984).

A range of empirical studies has been carried out in order to provide support for the notion of the programming plan and to establish the relationship of such plans to expertise in programming (Bonar & Soloway, 1985; Detienne & Soloway, 1989; Soloway & Ehrlich, 1984). For example, Soloway and Ehrlich found that experts were significantly more likely to complete gaps in programs with appropriate planlike constructs than were novices. In
studies of program recall, experts tend to recall those elements of a program that represent particular plans before they recall other, less salient, elements of a program (Soloway & Ehrlich, 1984).

1.2 The Problems and Limitations of Plan-Based Approaches to Programming

1.2.1 Plans as Explanatory Mechanisms. Recent work in the psychology of programming has questioned the adequacy of the plan concept as an explanatory mechanism. In particular, a number of authors have sought to suggest that programming plans are limited in their range of applicability. For example, Gilmore and Green (1988), in a study in which they perceptually cued various information structures in programs (including plan structures and control flow, etc.), found that program plan structures are psychologically meaningful to Pascal programmers, whereas BASIC programmers tend to be influenced rather more by cues to control-flow in programs. Gilmore (1986) and Green (1989) used the term “role-expressiveness” to explain these language differences. They claimed that Pascal programs (with a greater variety of syntactic constructs) are in general more discriminable from each other than BASIC programs, and this, they suggested, allows the programmer to infer the role of a particular statement more easily.

Davies (1990a) put forward an alternative explanation for these language differences based upon findings which suggest that the design knowledge possessed by programmers plays an important role in the eventual formation and use of plan structures. This alternative explanation is based upon the observation that, in general, Pascal programmers will be more likely to have design experience than BASIC programmers. This is because Pascal is mainly taught in conjunction with various design and structured-programming techniques, whereas BASIC is more commonly taught in isolation as an adjunct to other disciplines. It is argued that this differential design experience may lead to differences in the way in which plan structures are perceived and used. The validity of this observation has been explored experimentally, and this has provided some support for the idea that design knowledge can affect the perception and use of plan structures (Davies, 1990a).

Here, design-experienced and nondesign-experienced programmers of equal competence or expertise (measured in terms of their exposure to the language and their general debugging skills) demonstrated a marked differential use of plan cues despite the programming language used. This finding has been interpreted as demonstrating that although novices may experience some difficulty learning to express plans, they can benefit from training in design. This is not because design training involves the explicit teaching of such plans. Rather, knowledge about the design process appears to provide a means of facilitating the selection of the salient features of plans, and
enhances the mapping between structures in the problem domain and structures in the language domain. In a similar vein, Stone, Jordan, and Wright (1990) provided evidence that learning structured programming through Pascal education can improve programmers' general debugging performance by increasing the comprehension of program goals and plans.

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

1.2.2 Strategy versus Knowledge. Also, in addition to questioning the assumed universality of the programming plan, the work previously cited suggests other problems with the plan theory of programming. Bellamy and Gilmore (1990) compared the coding behavior of experienced programmers using a number of different languages. The intention of their work was to examine whether the order of program generation suggested the existence of planlike structures. Their evidence for the use of plan structures in program generation was equivocal. Hence, the question that arises is why plans should prevail in the comprehension process (as in Soloway's recall experiments) but not during generation? This concurs with Pennington's (1987) contention that "While plan knowledge may be implicated in some phases of understanding and answering questions about programs, the relations embodied in the proposed plans do not appear to form the organizing principles for memory structures" (p. 327).

It may be that the appearance of plan-based behavior is determined by comprehension strategy rather than knowledge. Hence studies, which have examined recall as opposed to generation, may, as Bellamy and Gilmore suggested, have tapped post-hoc rationalizations of the programmer's behavior. With this in mind, it would seem reasonable that studies investigating such issues should aim to clarify or make explicit the particular role of strategy versus knowledge in the determination of programming behavior (Gilmore, 1990). In addition, it is clear that studies examining program recall should not be assumed necessarily to be tapping the same knowledge structures or programming strategies as those found to exist in code generation.

1.2.3 Addressing Strategy. Other problems with the plan notion are that little attention has been paid to a consideration of the mechanisms, which control plan selection or implementation, to the nature of the development of plans with expertise or to the dynamic aspects of plan use. Rist (1986, 1989) advanced a plan-based model of the programming activity which suggests that novices and experts employ very different strategies when developing a program to solve a particular problem. Rist claimed that expertise is
characterized by the ability of the programmer to focus upon the most salient parts of the plans comprising a program. Rist termed these the "focal lines" of the plans. Rist suggested that as expertise develops, some plans are automated (such as input and output) and initially ignored during design, thus enabling the programmer to direct attention to the more difficult or novel segments of code. In terms of Rist's framework, as expertise increases, plans are selected rather than constructed, and knowledge of the plan focus reflects this increase in expertise.

In contrast to Rist's model, Green, Bellamy, and Parker (1987) suggested that when code is not generated in a strict linear fashion (which they claimed is the "natural" development path for the construction of programs; cf. Hoc, 1981), it is primarily because of problems with notational features of the programming language (partly because of its limited role-expressiveness) or because of constraints imposed by the so-called "device language." Following Payne (1987) and Payne, Squibb, and Howes (1990) in this context, the term, device language, is employed to refer to the language the user invokes to give commands to the text editor or its equivalent.

In addition to these two determinants of strategy, Green et al. (1987), also suggested the importance of the programmer's knowledge representation, but in contrast to previous studies, which emphasized the role of knowledge representation, their empirical work focused upon an investigation of those features of both the task (programming) language and the device language which are thought to determine strategy.

1.3 Towards an Integrated Developmental Framework for Understanding Programming Behavior

It is likely that features of the device language, the task language, and the programmer's knowledge representation interact to determine the nature of programming strategy. The work reported in this article provides empirical support for a model that was constructed to explain the development of programming strategy and to clarify the nature of the interactions between programming language features and the development of those structures which are hypothesized to represent programming knowledge. The aim of this work is twofold. First, to provide additional support for the models of coding presented by Green et al. (1987) and by Rist (1986, 1989); and second, to extend and elaborate these models by exploring the way in which their separate aspects (i.e., Green et al.'s emphasis on task language features and the programming environment, and Rist's emphasis on knowledge representation) might be combined to form a single and unified developmental framework. The intention of this framework is to show how programming strategies change with changes in knowledge representation and to highlight the effects of the notational features of the task language on the development of these strategies.
In order to investigate the strategic aspects of the programming activity, the work reported here used a method similar to one originally devised by Green et al. to examine nonlinearities in the coding process. Green et al. used discontinuities or "jumps" in the generation of program text to indicate departures from linearity. For this purpose, a jump was defined as an editing action that was followed by moving the cursor to another location and performing another editing action. An extension to this method, which taps more directly the role of knowledge representation in the determination of programming strategy, involves examining departures from linearity within and between the program’s plan structures. This method has been used successfully to investigate the more general effects of programming language notation and skill differences on strategy (Bellamy & Gilmore, 1990, Davies, 1989, 1991). Essentially, this method involves identifying the plan structures in code that have been generated or reconstructed from memory, and then analyzing, from transcripts of the coding activity, the number of jumps made between lines within the same plan structure (intraplan jumps) and the number of jumps made between lines that form part of different plan structures (interplan jumps).

Such jumps can be characterized as the points at which programmers make some change in their code. Gray and Anderson (1987) introduced the notion of "change episodes" to describe these key junctures in the coding process. They suggest that change episodes can be implemented in two distinct ways: either as minor local amendments to a program, or as major transformations in the programs’ goal structure. The importance of this work is that it clearly identifies the circumstances in which such change episodes might occur; for instance, as an interruption to coding, in response to other change episodes, or as a by-product of symbolic execution.

In addition, this work highlights the distinction between progressive (working towards the goal state of the problem) and evaluative (evaluating an executed part of the problem solution) activities in problem solving (Allwood, 1984). Most studies of the programming activity have addressed issues relating to the progressive aspects of problem solving and have tended to ignore the evaluation process. One notable exception to this is the parsing/gnisrap model (described later) proposed by Green et al. (1987) where the dichotomy between progressive and evaluative problem solving is given prominence.

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

First, if plan structures constitute the underlying cognitive representation of a program and are not language dependent, but are instead related to the programmer's level of expertise, then clearly this will be reflected in differences in the strategic use of plan structures by programmers of different skill levels. Hence, if Rist's model of coding is adopted, expert programmers should display strong nonlinearities in code generation in comparison to novices. In terms of Rist's model, experts might be expected to insert the focal lines of certain plans when appropriate, and suspend or ignore full development of these plans to concentrate on more difficult sections of code. Conversely, for the novice, code generation should be a largely sequential activity.

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

A second measure, which provides information about the factors that affect or contribute to programming strategy, is pause data. Such data has been used to indicate the independence of discrete plan structures in memory. For instance, Haberlandt (1980) found reading-time evidence for story episodes as independent memory units. Specifically, Haberlandt found that readers paused for a greater length of time at the beginnings and ends of episodes in stories. In a similar way, Robertson and Black (1986) showed that pause time increases between hypothesized plan boundaries in a text-editing task. Similarly, Reitman and Rueger (1980) investigated the organization of programmer's knowledge representation using a free-recall technique backed up with collateral converging evidence obtained from structures induced from the pattern of recall pauses.

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

2. METHODS

2.1 Subjects
Thirty-six subjects were recruited for this experiment. These subjects were classified into three groups of equal size according to their programming expertise. The novice group consisted of first-year undergraduate computer science students, all of whom had attended a preliminary short course in Pascal. All subjects in this group had some knowledge of BASIC, although this was limited to experience acquired during ad hoc courses prior to their matriculation. None of these subjects expressed the feeling that they could claim any particular expertise in BASIC. Indeed, this prior screening of potential subjects excluded a number of subjects from this group because of their wide-ranging experience of BASIC and concomitant knowledge of the language. A second group of subjects was classified as intermediate. This group consisted of second- and final-year computer science undergraduates. Subjects in this group had completed two single-term courses in Pascal, and all had employed the language extensively in project work. All subjects in this group professed to being reasonably conversant with BASIC. Indeed, most of the subjects classified as intermediate had used BASIC quite extensively during the early stages of their course. A final group of expert programmers consisted of subjects drawn from a population of teachers of programming and professional programmers employed in industry. None of the subjects in this group had fewer than 3 years postdegree programming experience, whereas a number of members possessed over 10 years postdegree experience.

2.2 Materials/Experimental Programs
Subjects were asked to produce programs from natural language specifications of three problems. One of these problems was the "rainfall problem" (see Figures 1 and 2) used by Johnson and Soloway (1985). The second problem involved determining whether an integer supplied as data was a prime number or not. The third problem specification was concerned with the calculation of a maximum and a minimum value from a series of numeric keyboard inputs.
Noah needs to keep track of the rainfall in the Manchester area in order to determine when he should launch his arc. Write a program which he can employ to do this. Your program should read in the rainfall for each day, stopping when the value 99999 is typed in. This value is not a date, but will indicate the end of input. If the user types in a negative value the program should reject it, since negative rainfall is not possible. Your program should print out the number of valid days that have been typed in, the number of rainy days, the average rainfall per day over the period, and the maximum amount of rainfall that fell on any one day.

**Figure 1.** A specification for the ‘rainfall problem’ used by Johnson and Soloway (1985).

```
PROGRAM Rain (INPUT, OUTPUT);
CONST STOP = 99999;
VAR Sum, Rain, Max, Ave: REAL;
Valid, Rainy, Dry: INTEGER;
BEGIN
  Sum := 0;
  Dry := 0;
  Rainy := 0;
  Max := 0;
  Write ('Enter rainfall: ');
  Readln;
  Read (Rain);
  WHILE Rain <= 0 DO
    BEGIN
      Write ('Rain: ', Rain: 0: 2, ' is not a possible rainfall, please try again');
      Read (Rain);
    END;
  WHILE Rain <= STOP DO
    BEGIN
      Sum := Sum + Rain;
      IF Rain <= 0 THEN
        Dry := Dry + 1
      ELSE
        Rainy := Rainy + 1;
      END;
      IF Rain > Max THEN Max := Rain;
  Write ('Enter rainfall: ');
  Readln;
  Read (Rain);
  WHILE Rain <= 0 DO
    BEGIN
      Write ('Rain: ', Rain: 0: 2, ' is not a possible rainfall, please try again');
      Read (Rain);
    END;
  END;
END.
```

**Figure 2.** A standard Pascal solution to the rainfall problem with a number of plan structures illustrated (from Johnson and Soloway, 1985).
2.3 Design

The experiment employed a program-generation task. The two independent variables were (1) language (Pascal/BASIC) and (2) skill level (novice/intermediate/expert); the two dependent variables were (1) number of jumps performed by jump-type classification (inter/intraplan jumps), and (2) length of pause between jumps by jump-type classification (inter/intraplan jumps).

2.4 Procedure

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

A number of protocols were established in order to ensure a level of consistency within these different measures. The presence of plan structures was analyzed by a group of three experienced programmers, all of whom were briefed about the nature of programming plans and were informed which plans might be expected to occur in each of the programs. These plans were derived from Johnson and Soloway (1985). Each member of the group analyzed all the resulting program-generation transcripts in terms of the expected plan structure of the program. They were asked to associate each line of the program with a specific plan. The raters were requested to carry out their analysis in terms of the plans identified by the experimenter. However, they were encouraged to suggest other plans within the program that were not made explicit in the initial plan analysis. It should be noted that no new plans were identified during this process. Figure 3 shows two program fragments illustrating comparable structures in BASIC and Pascal.

Plan jumps were defined as follows: Intraplan jumps were classified as movements between a current cursor position to positions within the same plan structure. Interplan jumps were classified as movements between a current cursor position to positions within different plan structures. These protocols applied to situations where new text was being inserted or existing
BEGIN
Sum:=0;
Rain:=0;

Read(Rain);
WHILE Rain<>99999 DO
BEGIN
  IF Rain<0 THEN
  ELSE
    BEGIN
      IF Rain=0 THEN
        Varld:=Varld+1
      ELSE
        BEGIN
          Vald:=Vald+1
          Rainfall:=Rainfall+1
        END;
      Sum:=Sum+Rain
      IF Rain>Max THEN
        Max:=Rain
    END
  END
  Read(Rain);
END;
Average:=Sum/Varld;

a). A Pascal program fragment indicating a running total loop plan and an average plan

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

140 INPUT New
145 IF New=99999 THEN GOTO 170
150 LET Sum=Sum+New
160 LET Count=Count+1
170 UNTIL New=99999

190 IF Count=0 THEN PRINT "No legal input" ELSE PRINT "Average is..."; Sum/Count

b). A BASIC program fragment indicating a running total loop plan and an average plan

Figure 3. Two program fragments representing similar plan structures in Pascal and BASIC. Note that these programs do not compute exactly the same function. This reflects the variation typically found in the subject's answers.
text modified. Pause time between jumps was recorded in milliseconds, but this level of recording sensitivity was not thought necessary for the analysis. Hence, pause time is represented to the nearest second.

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

3. RESULTS

3.1 Plan Jumps

Figure 4 shows the mean number of inter- and intraplan jumps performed by subjects during generation by novice, intermediate, and expert programmers using either Pascal or BASIC. These data were entered into three-way
analyses of variance (ANOVA) with the following variables in each case: (1) skill-level (novice/intermediate/expert); (2) jump-type (inter/intraplan jump); and (3) language (Pascal/BASIC).

No main effect of skill level or language was apparent. There was a significant interaction between jump-type and skill level, $F(2, 132) = 6.34$, $p < .01$. This appears to reflect the orthogonal relationship between inter- and intraplan jumps with increasing levels of expertise. In addition, a complex three-way interaction among language, jump-type, and skill level was evident, $F(2, 66) = 3.72$, $p < .05$. Separate ANOVAs for the results from each jump-type classification were employed in an attempt to clarify the nature of this more complex interaction. These ANOVAs revealed that the Skill Level x Language interaction was significant in the case of the interplan classification, $F(2, 66) = 8.43$, $p < .01$, but not for the intraplan classification. This interaction appears to be a consequence of the greater number of interplan jumps performed by intermediate and expert Pascal programmers in comparison to their BASIC counterparts. There were no other significant main or interactional effects.

3.2 Pauses

Pause data (Figure 5) were analyzed in a similar fashion using a three-way ANOVA. The three variable levels were the same as before. There was no main effect of either language or skill level. The effect of jump-type was highly significant, $F(2, 132) = 18.2$, $p < .01$. There was a significant interaction between skill level and jump-type, $F(2, 66) = 5.89$, $p < .01$, and a three-way interaction among language, skill level, and jump-type, $F(2, 66) = 10.74$, $p < .01$. Again, separate ANOVAs for each jump-type classification were carried out. This procedure indicated that the Language x Skill Level interaction was significant in the case of the interplan classification, $F(2, 66) = 7.55$, $p < .01$, but nonsignificant for the intraplan classification.

3.3 Additional Language Comparisons

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

A measure of interjudge scoring reliability was obtained in order to ensure a level of concordance among the three judges assessing programs for the presence of plans. A high coefficient of concordance was found to exist among judges (Kendall’s coefficient of concordance $W = 77$, $p < .01$).
In addition, a plan analysis of all the resulting program transcripts was carried out by the experimenter in order to attempt to corroborate the results produced by the analysis. Both analyses were carried out independently and there was a high level of concordance between experimenter and judge's analyses both in terms of the plans that were identified ($W = .57$, $p < .01$) and their identification with particular lines of code in the program ($W = .64$, $p < .01$).

The program transcripts were also analyzed in terms of the comparative distribution of plans in Pascal and BASIC. This analysis was undertaken because it is possible that the plans generated in one language might be implemented in a localized group of statements, and in another language be more widely distributed in the program, thus giving rise to problems interpreting the plan-editing process.
Here, all lines in a program that corresponded to the same plan construct were identified. Each set of program statements (corresponding to a particular plan) were given the same label. Next, the distance from the first line of a particular plan to other lines comprising that plan were assessed. Hence, if the next line of the plan were immediately adjacent to the initial line, this was scored as zero, if it occurred on the next line it was scored as one, and so on, until reaching the last statement of a particular plan. This procedure provides a broad measure of the distribution of the elements of a particular plan. An average indication of plan distribution in the two languages can be computed by summing these distribution measures for each plan and dividing this by the total number of lines comprising each plan. The average distribution measure for Pascal plans (.11) did not differ significantly from the average distribution measure for BASIC plans (.09), [t test]. The fact that this distribution measure is greater than zero for both languages does suggest some plan distribution, however this distribution is minimal and, by and large, plans appear to be generated in spatially contiguous blocks of code both in Pascal and BASIC.

4. DISCUSSION

Taken together, these data suggest rather complex relationships among skill level, language, and programming strategy. Although they provide support for particular elements within existing theories of programming, they also reveal some of the more complex interactions among these elements which are not predicted by such theories. For instance, as predicted by Rist (1986), the results of the study here indicate changes in the strategic use of plans as skill levels increase. Hence, the linear generation model adopted by novices is replaced by a strategy that appears to reflect a development in plan focus as expertise increases. This is evidenced by the significant increase in inter-plan jumps with increasing expertise and a concomitant decrease in intra-plan jumps during the program-generation task. These data also support a view of program generation that is similar to Jeffries' (1982) analysis of the strategies involved in reading programs. Jeffries found that experts read programs in the order in which they would be executed. In contrast, novices tend to read programs from beginning to end, in linear order like a piece of text.

In addition, the pause data suggest that as expertise increases the time spent pausing between both inter- and intraplan jumps decreases. This may reflect the type of speed-up function typically found in other studies of skilled performance within both a general problem-solving context (Anderson, 1982) and also within the programming domain (Anderson, 1987; Davies, 1990b; Wiedenbeck, 1985). Also, evidence for plan boundaries is suggested
by the interaction between jump-type and skill level for pause data. Hence, the pause time between interplan jumps is greater than that between intra-plan jumps for all skill levels. As in other studies (Haberlandt, 1980; Reitman & Rueter, 1980; Robertson & Black, 1986) this suggests the existence of discrete plan boundaries in the programmer's knowledge representation.

The effects of language on generation strategy are, however, rather less straightforward. The model of coding presented by Green et al. (1987) predicts a clear relationship between the notational features of particular languages and the development of programming strategies. Hence, if this model were correct, a language such as BASIC should inhibit plan use because of certain features of its notation, whereas Pascal, which is thought to offer greater plan discriminability, should facilitate plan use. The results of this study, however, suggest that language has little overall effect upon programming strategy. This concurs with Bellamy and Gilmore's (1990) results, which also failed to provide evidence for a straightforward relationship between language and programming strategy.

Although the results of this study do not reveal a main effect of language, they do indicate a potentially interesting three-way interaction among jump-type, language, and skill level. Further analysis suggests that this interaction results from the fact that a greater number of interplan jumps are performed by intermediate and expert Pascal programmers in comparison to their BASIC counterparts. One reason for this seems to be that the effect of notation in the determination of programming strategy plays a greater role as programming skills develop, and particularly at intermediate skill levels.

Additional evidence for this interaction effect is to be found in the analysis of pause data during program generation. Once again, no main effect for language was evident, however the Language x Skill Level interaction was highly significant in the case of the interplan jump classification. From Figure 5 it is clear that, for intermediates, the length of pause between interplan jumps in Pascal is significantly less than that occurring between interplan jumps in BASIC. This may suggest that, as programming skill develops, the notation of the language (and its related plan discriminability, etc.) has a significant effect upon plan use, but is of little relevance as a determinant of strategy at lower and higher skill levels.

The interpretation of this effect may be quite straightforward. For the novice, plan use is hypothesized to be minimal (as demonstrated here), hence, one would not expect notational effects to play a role. As expertise increases then, the mean length of pause between interplan jumps for Pascal falls sharply. In contrast, there is only a slight reduction in the mean length of pause for the BASIC data. Hence, for Pascal programmers, at the earlier stages of skill development, the notation of the language appears to support the use of plan structures. This is evidenced by the reduction in the mean length of pause which is taken to be an indicator of the ease with which plan
DETERMINANTS OF PROGRAMMING STRATEGY

structures can be used (selected or implemented). The small decrease in the mean length of pause for BASIC programmers at early stages of skill development is assumed to support the contention that the notation of this language does not support the use of plans.

At later stages of skill development, the results indicate an interesting inversion of the preceding findings. Assuming a straightforward notational view, one would expect to find a linear relationship between the ability to use plans and increasing levels of expertise, regardless of language. However, Figure 5 illustrates that, for Pascal programmers, the rate at which the mean length of pause reduces between intermediate and expert skill levels is minimal. For BASIC programmers, however, there is a sharp decrease in the mean length of pause between these particular skill levels.

Once again, these results give credence to the idea that, at early levels of skill development, the notational aspects of Pascal support the use of plan structures, but once expertise has developed to a certain point, the effects of notation become less important. For BASIC, the development of the ability to use plan structures appears to be hindered during the beginning stages of skill development. However, the initial adverse effects of notation on plan use soon diminish as programmers reach sufficiently high levels of skill. Hence, the question that needs to be addressed is why plan use appears to be differentially affected by the notational aspects of the programming language at different skill levels.

Green et al. (1987) suggested that code generation involves two fundamental psychological processes: one in which the external structure (program code) is created from the internal (cognitive) structure which represents the problem requirements, and an inverse process in which this internal structure is recreated when necessary from the external structure. Green et al. called the second of these processes parsing and the first gnisrap (the reverse of parsing). They suggested that because of the limitations of working memory, programmers need to make use of an external medium (e.g., the VDU screen) as a temporary store for code fragments as they are generated. The aim of parsing is to recreate the original plan structure from these code fragments. In addition, features of the notation of the language may aid or hinder plan use.

Green and Borning (1990), attempted to model aspects of this parsing process using a generalized unification parser (Kay, 1985), which can not only handle elementary forms of program discourse analysis, but in addition, can be made responsive to aspects of the program's typographical layout or notational structure. This model predicts that the structure of particular programming languages will influence parsing difficulty when there are few lexical or display-based features to aid the parsing process.

The formal structure of the programming language is not the only determinant of strategy. According to Green et al., strategy is also affected by
the user's knowledge of how to perform external tasks, by their knowledge representation. However, no consideration is given to the way in which the notational features of the programming language and the programmer's knowledge representation might interact to determine strategy.

Green et al. claimed that their parsing/gnisrap model is a model of expert coding behavior, but the results of the study here suggest that expert programming strategies do not appear to be affected by notational features in the way that parsing/gnisrap model would predict. It appears from the data that the parsing/gnisrap model provides a reasonable interpretation of intermediate performance, where notation appears to influence programming strategy in the manner predicted. The model would, however, also predict a similar strategy to prevail during expert performance. If one assumes, not unreasonably, that intermediates and experts have a similarly constrained working-memory capacity, then the only other factor in the parsing/gnisrap model, excluding notation, that might influence strategy would be the programmer's knowledge representation. Hence, it appears that for experts, features of their knowledge representation take precedence over notation as a determinant of strategy.

Different notations are likely to facilitate parsing to a greater or a lesser extent (Green, 1989, 1990). Hence, in terms of the preceding interpretation, experts appear to be able to make use of some feature of their knowledge representation for programs, which enables them to parse for particular structures more readily. One way in which parsing might be facilitated is via the recognition of "beacons" that serve to indicate the presence of particular program components (Brooks, 1983). Brooks suggested that the comprehension of programs can be characterized as an iterative process of hypothesis, verification, and modification of hypotheses. According to Brooks, once specific hypotheses about the presence of particular operations or structures in programs have been formed, the programmer will then attempt to verify these against the program text. It is proposed that programmers do not study a program line-by-line, but instead, search selectively for certain key features that may indicate the presence of particular structures. Brooks called these key features beacons.

Wiedenbeck (1986a 1986b) demonstrated that experts can recall these key lines or beacons much better than novices. In addition, the presence of beacons in programs has been shown to facilitate program comprehension (Wiedenbeck & Scholtz, 1989). These studies suggest that expert programmers possess and are able to access a representation of programming knowledge which in some way reflects the saliency of these key lines.

The results of this study tend to support those findings and in addition, provide a means of examining the way in which programmers' knowledge representations may develop. These data also suggest some quite subtle interactions between developments in knowledge representation and general notational features of particular languages. Hence, as programming skill
develops, features of the programmer’s knowledge representation appear to reflect the increasing role of beacons, focal lines, and the like, in the program comprehension and generation process. Hence, intermediate programmers, whom one might suggest are still involved in the process of developing these particular features of their programming knowledge, are likely to be affected, to a greater extent than experts, by the ease with which they can parse existing program structures back into internal semantic representations. The results previously reported appear to support this argument and, while not providing a basis for the rejection of existing models, clearly indicate the need for a framework which can allow for the integration of a range of different models in order to explain the richness of this particular form of behavior.

5. GENERAL CONCLUSIONS

5.1 Changes in Programming Strategy with Expertise

This article has demonstrated that a range of factors may contribute to the determination of programming strategy. These factors include the development of particular features of the programmer’s knowledge representation and the way in which the notation of a language might facilitate the parsing of code structures into cognitive representations and vice versa. The results of this study provide evidence for changes in strategy that are associated with increasing expertise. In terms of existing work, there appear to be a number of ways to account for these differences in strategy.

First, one of the outstanding features of expertise in programming and in other domains (Flower & Hayes, 1980; Guindon, 1990; Hayes-Roth & Hayes-Roth, 1979; Schoenfeld, 1985), appears to be the particularly opportunistic nature of preferred cognitive strategy for programming and other tasks. Hence, the programming activity cannot be characterized as purely sequential; rather, it might be better construed as consisting of bouts of activity, each involving the creation of code fragments. These fragments are, in turn, continually reevaluated and modified in respect to the particular goal or subgoal currently under consideration. In addition, the development of code may be postponed at any time in order that the programmer might direct his or her attention to other goals or subgoals, possibly in response to the recognition of previously unforeseen interactions between code structures. Opportunistic strategies of this nature have been highlighted in a number of previous studies concerned both with program and software design (Guindon, 1989; Guindon, Krasner, & Curtis, 1987; Ratcliff & Siddiqi, 1985; Siddiqi, 1985; Visser, 1988) and the coding activity (Green et al., 1987). The strong nonlinearities observed here, which appear to characterize both expert and intermediate program-generation behavior, may provide additional support for this opportunistic view of the coding process.
An alternative view of the strategic changes, which appear to accompany increasing expertise, might be to suggest that programming strategy is transformed from a depth-first novice strategy to a breadth-first expert strategy. This position was advanced by Jeffries, Turner, Polson, and Atwood (1981) who claimed that novices adopt a depth-first approach to problem solving in programming, that is, they tend to expand only one part of a solution at progressive levels of detail. In contrast, experts tend to develop synchronously many subgoals at the same level of abstraction before moving on to a lower level.

Although the results of this study provide clear evidence that programming strategy changes with increasing expertise, they do not provide a means of distinguishing a depth-first versus breadth-first view of the programming activity, such as that suggested by Jeffries et al. (1981), from an opportunistic characterization of expert programming strategy. However, this study does indicate various factors which appear to contribute to the adoption of such strategies or to the ease with which they might be supported. The following framework suggests two central determinants of programming strategy and stresses the fundamental importance of their interaction in the development of strategy.

5.2 The Role of Knowledge Representation in the Determination of Strategy
First, it seems clear that the programmer’s knowledge representation must support the representation of salient code structures. Such structures, which might be characterized as beacons or focal lines, act as partial descriptions of particular code fragments and provide reminders that a segment of a program may need completing at a subsequent stage. In addition, the development of these code structures appears to coincide with increasing expertise. This may suggest that as expertise develops, knowledge structures change such that the organization of these structures reflects the increasing importance of focal lines and beacons, and so on. Similarly, in the domain of software design, Jeffries et al. (1981) found that although novices used the same general problem-solving methods as experts, they lacked skills in two areas: applying processes for solving subproblems, and effective ways of representing knowledge. Jeffries et al. attributed this latter deficit to the inadequacy of the organizing functions provided by the novice’s immature design schemata.

Although this article does not seek to deny the existence of generic declarative representations of programming knowledge (i.e., programming plans), it does suggest the need to consider the development of an asymmetry in programmers’ knowledge structures with increasing expertise. Hence, such knowledge structures appear to facilitate the representation of the focal aspects of plans, whereas the necessary, yet minor and subordinate parts of
plans are represented with less saliency. Here, one might the adopt the common view of natural language text comprehension (Bower, Black, & Turner, 1979; Kintsch 1974; Rumelhart, 1975) and conceive of programming knowledge as being represented in terms of hierarchically structured schemata, with focal plan elements achieving prominence in each plan or schemata hierarchy.

This view clearly does not rule out a plan-based approach, however, it does suggest certain limitations for the plan theory of programming. One implication of this study is that it would appear that the plan theory does not provide an adequate basis for a theory of programming expertise. Hence, the expert programmer does not simply have more plans than the intermediate. Rather, the development of expertise might be better characterized as a "fine-tuning" activity whereby the focal elements of plans are identified, and the kinds of knowledge asymmetry previously discussed are established.

It is the case that novice programmers seem to be able to access fewer plans than both intermediates and experts. This may mean that the plans novices are able to access are poor matches to the current goal under consideration. Hence, novices may feel the need to correct these badly matching plans immediately. This may give rise to the finding that novices tend to work on one plan at a time and to the associated prevalence of intraplan jumps that has been observed in the context of novice behavior. Conversely, experts, who have a greater range of plans may be able to find better matches to a current goal and hence may be more prepared to suspend the development of a particular plan until later. This view of the development of programming expertise would suggest that programmers gradually acquire program-specific plan constructs, and as a consequence, one would presumably expect intermediates to possess a greater range of plans than novices and a correspondingly smaller range than experts. However, the results of this study indicate that experts and intermediates generate the same number and range of plans, whereas novices, in comparison, generate significantly fewer plans. A similar finding has been observed in other analyses of plan generation by novice, intermediate, and expert programmers (Davies, 1989). Additionally, intermediate and expert programmers appear to perform at the same level when asked to detect plan violations in programs, suggesting a similar level of plan knowledge. Novices, by contrast, are very poor at detecting plan violations (Davies, 1990b).

In the present context, one might suggest that, although both intermediates and experts have access to the same number and range of plans, in the latter case, the representation of these plans has become attenuated in order to reflect a growing recognition of the importance of the focal aspects of these plans. A model of programming knowledge such as that advanced by Rist (1986, 1989), which represents plan schema in terms of frame and slot mechanisms, may provide an account for this knowledge asymmetry. Here,
each plan schema is denoted by a frame that represents the purpose of the schema, the role of its components as represented in code, and its pre- and postconditions. The frame identifier represents an instantiation of the focal line of the plan schema.

The results of this study provide support for this kind of representation of programming knowledge, and, in addition, suggest some clear parallels with studies of knowledge representation in the text-comprehension domain. This study also contributes to the understanding of the processes that underpin schema development, and provides a framework for elaborating the relationship between the development of knowledge structures and expertise. This is likely to have implications for schema theory as it is applied in other domains, because few studies have been concerned with the detailed relationship between schemata development and expertise. Where this relationship has been studied, expertise differences are normally explained in terms either of the relative completeness or incompleteness of schemata (Lesgold et al., 1988), or in terms of the presence or absence of particular schemata (Soloway, Adelson, & Ehrlich, 1988). In the context here, emphasis is placed upon the way in which knowledge is structured rather than upon the presence of schemata or their relative completeness (cf. Lewis, 1981; Rumelhart & Norman, 1978).

5.3 Notation as a Determinant of Strategy
The second part of the framework here is concerned with an analysis of the ways in which programming language notation might support programming strategy. One of the more interesting findings of this study is that notation does not appear to support an opportunistic or a breadth-first strategy to the same degree for programmers of different skill levels. That is, the effects of notation on strategy are less extensive for experts than for intermediates. In terms of existing theory (Green et al., 1987), this effect would not be predicted, because there is no reason to believe that features of the task-language notation should provide differential support for programming strategy, regardless of a programmer’s level of expertise.

One way to explain this differential effect might be to suggest that notation and knowledge representation interact very strongly to determine strategy. Hence, as representations of programming knowledge are in the process of development, as suggested in the case of intermediates, then any additional means of facilitating programming strategy, such as might be provided by certain features of the notation, are likely to be of particular importance. At higher levels of skill, factors relating to the organization of knowledge appear to play a greater part in the determination and the support of programming strategy.

In addition, the work reported here suggests a slightly different model of planning to those models which are generally advanced as descriptions of
the human planning activity. Most extant models of planning embody some abstract notion of the planning activity in which the nature of the problem representation is not shown to have an effect upon the kind of planning or problem-solving strategy that is invoked. However, within the programming domain it is clear that features of the notation of the task language can facilitate, or indeed act to constrain, the preferred cognitive strategy that is adopted for this task. This is likely to have more general implications for the study of planning within the range of domains that use formal or semi-formal notations to describe aspects of the problem space. As a consequence, it might be suggested that there are certain dangers in attempting to divorce planning models from an analysis of the way in which tasks might be represented.

6. SUMMARY

The framework advanced here suggests a number of implications for the understanding of the determinants of programming strategy. In particular, it is clear that the role both of notation and knowledge representation cannot be fully explained in isolation. This is because these factors appear to interact very strongly to determine programming behavior. Hence, it is only through an analysis of these more complex interactions that a comprehensive elaboration of the determinants of programming strategy will be forthcoming. This article suggests the need to consider these interactions within a developmental framework. That is, within a context that views the development of programming skill as accompanied by subtle changes in the way in which programming knowledge is represented. This fine tuning of programming knowledge appears to provide support for particular forms of preferred programming strategy. The use of such strategies is, in turn, assisted by features of the language notation, but only during the beginning stages of their development. As programming skill increases, the role of notation appears to take less precedence as a determinant of strategy.

This analysis suggests that previous studies, which have examined the nature of programming strategy, only provide an interpretive basis within a rather limited context. Hence, such studies, although emphasizing the need to consider both notation and knowledge representation, have failed to elaborate the relationships and interactions among these factors within the general realm of skill development in programming. This article attempts to build upon the results of these studies, while at the same time stressing the fundamental nature of the interactions among these central determinants of strategy. A comprehensive understanding of the strategies involved in a complex task such as programming is only likely to be facilitated if one both delineates the role of its individual components, and also understands in detail the intricate nature of their interactions.
REFERENCES


