THE QUARTERLY JOURNAL OF EXPERIMENTAL PSYCHOLOGY, 1988 *40A* (3) 423–442

Programming Plans and Programming Expertise

D. J. Gilmore
University of Nottingham, U.K.

T. R. G. Green
MRC Applied Psychology Unit, Cambridge, U.K.

This paper addresses issues of the nature of expertise in programming and asks whether "programming plans" represent the underlying deep structure of a program. It reports an experiment that investigated the effect, on experienced programmers, of highlighting the plan structure of a computer program, while they were performing both plan-related and unrelated tasks. The effect was examined in both Pascal and BASIC. For Pascal programmers, perceptual cues to the plan structure were useful only for plan-related tasks, but the same cues were of no benefit to experienced BASIC programmers in any of the tasks. These results suggest that the actual content of programming plans does not generalise across different languages, although it is possible that the BASIC programmers can use other plans. From these results a more detailed description of programming plans and their role in programming expertise can be developed. The fact that BASIC programmers were not sensitive to the same plans as Pascal programmers implies that plans cannot represent the underlying deep structure of the programming problem.

The "programming plan" has been proposed by Spohrer, Soloway, and Pope (1985) and by Rist (1986) as a major feature of programming expertise. Expert programmers are assumed to acquire a repertoire of such plans, which represent stereotypic code fragments, allowing them to generate code and recognise its structure more easily. Programming plans have been used as an important concept in the development of programming tutors (e.g.

Requests for reprints should be sent to D. J. Gilmore, Psychology Department, University of Nottingham, Nottingham NG7 2RD, U.K.

We are extremely grateful to all those who helped us at the various stages of this research, but particularly to the various departments at Cambridge, Sheffield, and Lancaster who supplied the programmers and to John Self at Lancaster for allowing time to complete the experiment.

EP A 40/3-A

Bridge: Bonar & Cunningham, in press) and in teaching aids, such as debugging tools (e.g. Proust: Johnson, 1986). However, despite this widespread acceptance of programming plans, there are a number of important unresolved issues. For example, the psychological nature of plans has not been adequately described, nor has the generality of plans (to other programming languages) been discussed.

What are Programming Plans?

The theory of programming plans is to describe the errors that novice programmers make and to explore the misconceptions that underlie them. It is intended that the model should describe the process by which these misconceptions are corrected. The theory claims that expert programmers develop both more plans and higher-level plans than novices, and that this acquisition of plans is the characteristic of expertise (Rist, 1985). However, before submitting the theory to empirical test, it is necessary to be clear exactly what is meant by the term "programming plan".

The analysis derives from the structured programming philosophy, which solves a problem through a process of top-down refinement. Sub-goals that are generated by this process can either be treated as problems to be solved, or else a solution may be immediately available to the programmer. In the latter case the programmer could be said to have a plan for that goal. The added complexity is that the plans must not only be generated, but they must also be interleaved with each other in order to produce a correct program. The theory also builds on research from the domain of text comprehension, which has provided empirical support for the cognitive reality of goal-based and plan-based knowledge (Bower, Black, & Turner, 1979; Schank & Abelson, 1977).

Rist (1985) has attempted to formalise the concept of a programming plan and has described how it underpins the transition from novice to expert. He describes a collection of different sorts of plans, all represented as slot-and-filler mechanisms. Program plans—PPlans—(see Figure 1) are basic plans similar to those described by Soloway and Ehrlich (1983). They have slots for the goal of the plan, the code generated by the plan, etc. Plans for counting, summing, and searching are of this type.

Complex program plans—CPPlans—are constructed from a number of PPlans, in order to achieve goals at a higher level. Other types of plans are abstract plans (APlans), which represent knowledge of different types of loop, different types of sort etc., specific plans (SPlans), which represent specific routines, such as bubble sort, and global plans (GPlans), such as "initialize", "validate" or "update". Although such knowledge undeniably exists and is understood by experts, the distinctions between these different types of plans are not very clear. For the purposes of this paper, therefore,

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

Variations:
i) loop using ?found as end test
ii) loop using ?found within the loop
iii) a marker to be used outside the loop

Included plans: None

Optional plans: Store ?x when found

PPlan - 'found' plan.

FIG. 1 Plan descriptions (from Rist, 1985).

PPlans will be assumed to be the typical plans, as they represent the point of agreement across all the writing on programming plans. In particular, the plans used in the experiment reported below are the "average", "filter", "input" and "maximum" plans.

Empirical Studies

The majority of studies addressing the content and structure of expertise in programming have been conducted by the Cognition and Programming Project at Yale. A number of experiments has been conducted to demonstrate the possession of such plan knowledge by expert programmers.

Soloway, Ehrlich, Bonar, and Greenspan (1982) and Soloway and Ehrlich (1983) have presented experimental evidence for the ability of expert programmers to use plans. For example, in one study novices and experts were required to complete the initialization statements at the beginning of a program. Experts were significantly more likely to fill the gap with appropriate constructs than were novices, suggesting that they have acquired plan knowledge about the roles of variables and forms of initialisation and update. Other studies have shown that novices frequently make errors as a result of attempts to achieve two goals with a single plan (referred to as "merged goals").

Rist (1985) collected both experimental and protocol data to investigate the development of plans and expertise. In his experiment (with 11 novices and 15 experts) the programs were analysed into goal and plan structures, the former being at a higher level than the latter. Hypothesizing that the higher-level structure would be more beneficial to the experts, Rist gave both types of information to groups of novices and experts, expecting an interaction. Instead, the plan structure proved most useful to both groups in a debugging task. Rist argues that this is not evidence against the theory, but that it is due to his failure to analyse the plan and goal structure of the program correctly.

The protocols, obtained from novices and experts, provide a clear demonstration of the use of plans by experts. For example, when comprehension behaviour was classified as either deductive or inductive, Rist observed that experts generally deduced the presence of plans and matched the code against their deductions, whereas novices induced and constructed plans while reading the code. Unfortunately much of this analysis is subjective, with Rist consistently using examples from the performance of the "much less skilled" novice (there were only two novices).

Problems with Programming Plans

The above analysis of programming plans raises three important questions, some of which have already been mentioned:

1. *Are plans psychologically real?* Their existence is mainly inferred from protocol and error analysis, rather than from direct experimental evidence. Such support can be obtained by showing that perceptual cues to the plan structure of a program improve the comprehensibility of the program.
2. *Are plans the main psychological representation of programming knowledge?* The plan theory suggests that plans are the expert programmer's mental representation of a program, that they represent the deep structure of the problem. Thus, providing a situation in which the plans can be readily perceived should improve performance on all programming tasks.
3. *Are the observed plans related to the problem being solved, or the language being used?* All the existing evidence comes from studies with Pascal programmes, with the assumption that the effects will generalize to other languages. But there have been no attempts to examine whether programmers in other languages use similar plans.

The experiment to be described addresses these three questions by providing perceptual cues to different structures in a program and observing those tasks in which performance is improved. Thus, one can discover which

tasks require information from which structures and which structures are understood by the programmer (cf. Gilmore & Green, 1984; Gilmore, 1986a). In the experiment the tasks compared are the detection of a variety of bug-types, with perceptual cues provided to both plan and control structures.

Classifying Errors Through Plan Analysis

Spohrer et al. (1985) analysed a large collection of novice Pascal errors (Johnson, Soloway, Cutler & Draper, 1983) according to the plan theory. Their classification scheme describes bugs as differences between a novice's implementation of a plan and a correct implementation of that plan. The difference is described in terms of the component of the plan in which is occurs (for example, input, initializations), and within these components the difference can be described as either missing, malformed, spurious, or misplaced.

However, there are problems with this scheme because it is what Reason (1984) describes as a behavioural scheme, relying upon simple, observable categories such as omission, substitution etc. In the experiment described below, a different classification scheme is used, which does not examine discrepancies in the behaviour of the correct and the buggy program, but in their structure. The following categories of bugs are distinguished:

1. *Surface level bugs.* These bugs are independent of any particular structure in the program and may be caused by typing errors and syntactic slips (for example, missing or misplaced quotes or undeclared variables).
2. *Control-flow bugs.* These occur within the control-flow structure of the program, without affecting other structures. They may occur in one piece of control-flow, or at the interaction of two bits of control-flow (for example, a missing "begin" statement).
3. *Plan structure bugs.* Even when the control-flow structure is correct, the plan structure may contain errors (for example, updating the wrong variable).
4. *Structure interaction bugs.* Both the control-flow structure and the plan structure may be correct, but the interaction of the two may contain errors (for example, initializations within the main loop).

Making Structure Apparent

Gilmore and Green (1984) and Gilmore (1986a) have shown how providing perceptual cues to aspects of program structure can lead to large improvements in the performance of relevant programming tasks. The issue of whether programming problems possess an underlying deep structure gives rise to the specific question of whether providing perceptual cues to the plan

structure of a program improves specific task performance or general comprehensibility. Given the above bug classification scheme, it is appropriate to compare the highlighting of plan structures with the highlighting of control-flow structures.

Highlighting Control-flow Structure—Indentation

Many experiments have been performed on the value of indentation in programs (e.g. Miara, Musselman, Navarro, & Shneiderman, 1983; Kesler et al., 1984) with apparently conflicting results. But a closer inspection reveals that they are in agreement: indentation is useful for tasks that require an understanding of the control-flow structure of the program, but it is not useful for other programming tasks. Thus, indentation seems like the ideal perceptual cue to use to highlight the control-flow structure of the program without affecting the perception of other structures.

Highlighting Plan Structure—Colour

If indentation were used to highlight plan structures as well, then it would be impossible to construct an experimental condition in which both types of structure were cued. Thus, the perceptual cues to plan structure must not conflict with indentation. This can be achieved by the use of colour cues, in which fluorescent highlighting pens are used to group the lines of code which belong to the same plan.

Hypotheses

The results of Gilmore and Green (1984) suggest that it is reasonable to expect that plans are just another view of the program code, rather than a mental language of programming, and claims by Anderson (1985) suggest that plans will be equally useful to expert BASIC programmers as to expert Pascal programmers. Thus, the following predictions can be made:

1. *For surface errors:* The presence of cues will make no difference to the detection of surface errors.
2. *For control-flow errors:* Indentation cues to control-flow will improve the detection of control-flow errors (i.e. no cues and plan structure cues will produce a lower detection rate than control structure cues or both cues).
3. *For plan errors:* The use of colour cues to plan structures will improve the detection of plan errors (i.e. no cues and control structure cues will produce a lower detection rate than plan structure cues or both cues).
4. *For interaction errors:* The presence of both types of cues will

improve the detection of these errors (i.e. both cues will produce a higher detection rate than the other three conditions).

This last prediction is less certain than the others, and it is included more for the sake of completeness, as there is no evidence to suggest how programmers will react to the presence of two types of perceptual cue in the same program.

Method

Subjects. Subjects were recruited from final-year Computer Science undergraduates at the Universities of Cambridge, Sheffield, and Lancaster, from final-year Applied Maths students at Cambridge, and final-year Engineering students at Lancaster. There were 32 experienced Pascal programmers and 32 experienced BASIC programmers. The subjects were predominantly male, with an average age of approximately 22. For most students (44) the experiment occurred after their final examinations, whereas for a few (20) it occurred near the beginning of their final year. The experiment had no bearing on their course marks and subjects were paid for their participation.

Experimental Programs. Three problems were used, similar to those studied by the Yale project. One calculated an average over 10 inputs and was used as a practice problem (Problem 1). The other problems were more complex, one calculating an average over a certain number of filtered inputs (Program 2), and the other calculating the maximum as well (Problem 3). Buggy programs for each problem were created (four for the practice problem, ten each for the other two), and each buggy program contained two bugs, giving a total of 40 bugs in the experimental programs (ten of each of the four types).

Indentation was added to the Pascal programs by including 4 extra spaces following each *begin*, and removing these spaces before each *end*. In the BASIC programs, indentation was added in a manner that reflected the underlying control structure. The cues to the plan structures were introduced to both languages using fluorescent marking pens in three distinctive colours. In all programs the same colour was always used for the same plan. The plans involved were *input*, *average*, *filter* and *maximum*, but only three occurred in any one program.

Design. The experiment used a bug-detection task with four different program formats within each language:

1. no perceptual cues (no-cues);
2. perceptual cues to control-flow structure (control-cues);

3. perceptual cues to plan structure (plan-cues);
4. perceptual cues to both control-flow and plan structure (both-cues).

An example of the both-cues condition is given in Figure 2, in both Pascal and BASIC, although different fonts are used instead of different colours. Likewise, there were four bug types:

1. Surface errors;
2. Control-flow errors;
3. Plan structure errors;
4. Plan structure × Control-flow interaction errors.

The same bugs were used in all the format conditions, and as far as possible in both languages. Thus, the experiment used a mixed design, with two between-subjects factors (language and program format) and one within-subjects factor (bug type), with 8 subjects in each group.

Performance was measured by the number of each type of bug detected in each of the four program conditions. Figure 2a contains a Plan Error and an Interaction Error, and Figure 2b contains a Surface Error and a Control Error.

Procedure. The experimental programs were presented to subjects in a booklet, with one program per page. The practice problems occurred in a fixed order, but the order of the remaining programs was randomized, except that the filtering programs were presented before the maximum programs.

The initial instructions to the subjects described their task as one of marking novice programs that had been produced under examination conditions. This provided a justification for the task as a whole and for the repetition of a single program ten times. This description of the task also emphasized to subjects that their task was to *mark* the errors and *not to correct* them. Subjects were given a limited amount of time to mark each program (60, 80, and 100 sec on problems 1, 2, and 3, respectively).

Following these general instructions, subjects were given a specification of Problem 1—the practice problem—and an example correct program. They had 2 min to study this. The experimenter was available to answer any questions throughout this period. Next the subjects marked the four buggy practice programs. They were informed of the "expected" bugs after each program. Any alternative bugs suggested by the subjects were marked as well. The same routine was then followed for Problems 2 and 3, except that 3 min was allowed for the initial study of the problem specification and sample program, and the randomized order of the programs precluded the provision of feedback.

For each program subjects were told to look for errors, but not how many

```
10  program prob12;
20   vars depth, days, rainfall:integer;
30        average:real;

40  begin
50       for days := 1 to 40 do
60       begin
70          depth := 0;

80            writeln("Noah, please enter todays rainfall
90           readln(rainfall);

100         rainfall:= rainfall + depth;

110      end;
120     average := depth/40;
130     writeln("Average is ", average);

140  end.
```

(a) Pascal program (with plan error line 100 and interaction error line 70).

```
10  REM prob14
20  for n= 1 to 40
30       print "Noah, please enter todays rainfall"
40       read rain
50      total = total + rain
60      average = total /40
70      print "Average is avera"

80  next
```

(b) Basic program (with surface error line 70 and control-flow error lines 55/80).

FIG. 2 Example programs used in the experiment. Different fonts are used to represent the colour highlighting of different plans.

to look for. For each one they found they were asked to note the line number of the error and a short description of the error.

Results

The mean error detection rates are shown in Table 1. A three-factor ANOVA was performed on this data, revealing main effects of language, $F(1, 56) = 7.3$, $p < 0.01$, and bug type, $F(3, 168) = 26.6$, $p < 0.01$. The Language × Bug type interaction was significant, $F(3, 168) = 7.7$, $p < 0.01$, as was the three-way interaction between language, bug type, and cue type, $F(9, 168) = 2.2$,

$p < 0.05$. The emphasis of the ensuing analysis was to tease apart this three-way interaction, as simple interpretations of the other effects are not possible.

Separate ANOVAs for the results from each language revealed that the cue type and bug type interaction was significant for the Pascal programmers, $F(9, 84) = 2.7$, $p < 0.05$, but not for the BASIC programmers, $F(9, 84) = 0.05$. This interaction for the Pascal programs is illustrated in Figure 3, by showing performance in the three-cued conditions relative to the no-cues baseline. Planned comparisons were performed from the Pascal ANOVA to investigate whether performance was best when cue type and bug type were matched. These comparisons correspond to the four hypotheses presented earlier, except that they applied to both languages, rather than to Pascal alone. Thus, in relation to Hypothesis 1, for the surface errors an analysis of simple main effects revealed no effect of cues, $F(3, 84) = 0.02$. Hypothesis 2 concerned the plan structure errors, for which the comparison between plan cue conditions and the other cue conditions was significant, $F_{comp}(1, 84) = 9.8$, $p < 0.01$. For control structure errors (Hypothesis 3) the comparison between control cue conditions and the others was significant, $F_{comp}(1, 84) = 8.6$, $p < 0.01$. Likewise, the comparison for interaction errors (Hypothesis 4) between the both-cues condition and the other three was significant ($F_{comp}(1, 84) = 5.8$, $p < 0.05$). Thus, for the Pascal programmers,

TABLE 1
Error detection rates (percent) by error type and programming language.

	Error Types			
	Surface	*Plan Structure*	*Control Structure*	*Interaction*
PASCAL				
No Cues	36	67	46	60
Indentation	41	64	67	56
Colour	37	85	40	59
Both	41	81	51	74
Mean	39	74	51	62
BASIC				
No Cues	62	71	61	65
Indentation	67	82	62	66
Colour	59	67	64	60
Both	59	77	61	56
Mean	61	74	62	62

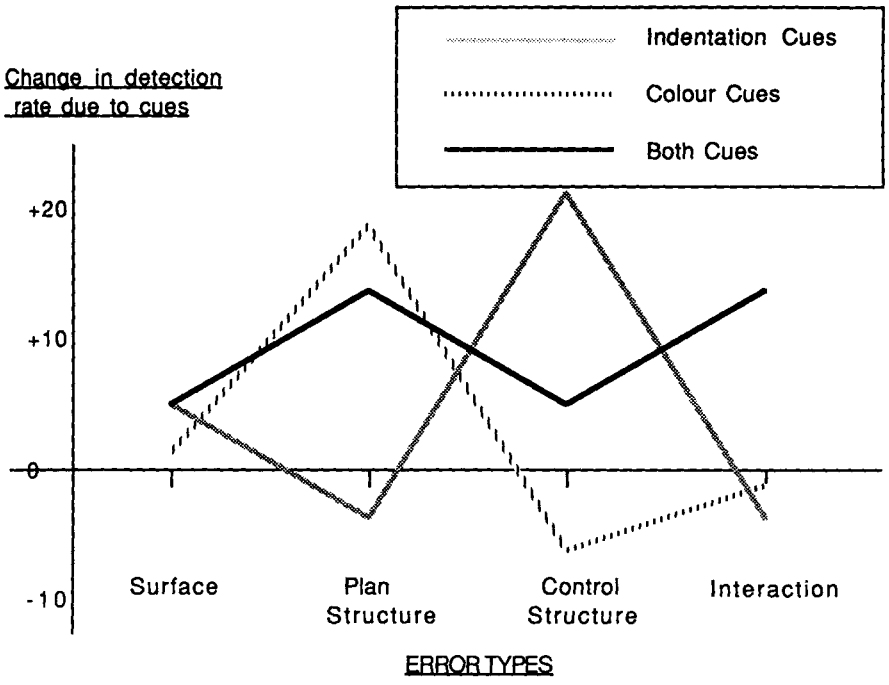

FIG. 3 Change in the error detection rates for the Pascal programs caused by the presence of cues.

cues to particular structures only improve the detection of bugs within those structures.

Figure 4 similarly shows the relative error detection rates for the BASIC programs, which are clearly different from those for the Pascal programs and from those predicted. The simple effects of cues were not significant for any of the four error types, $F(3, 84) = 1.5$ for Plan Structure errors and $F(3, 84) < 1$ for the other error types. Thus, the results for the BASIC programmers do not show any advantage for plan cues or for control cues.

The other effects observed in the three-factor ANOVA are closely related to the above analysis of the three-way interaction. The main effect of Language and the Language × Bug interaction cannot be easily interpreted, because the effects are different for the different cues. Appendix 1 presents a table of the simple effects of language, which reveals that only for surface errors did the BASIC programmers consistently detect more errors than the Pascal programmers, $F(1, 168) > 5$, $p < 0.05$), and Table 1 clearly shows the BASIC programmers were no better, on average, at detecting plan or interaction errors.

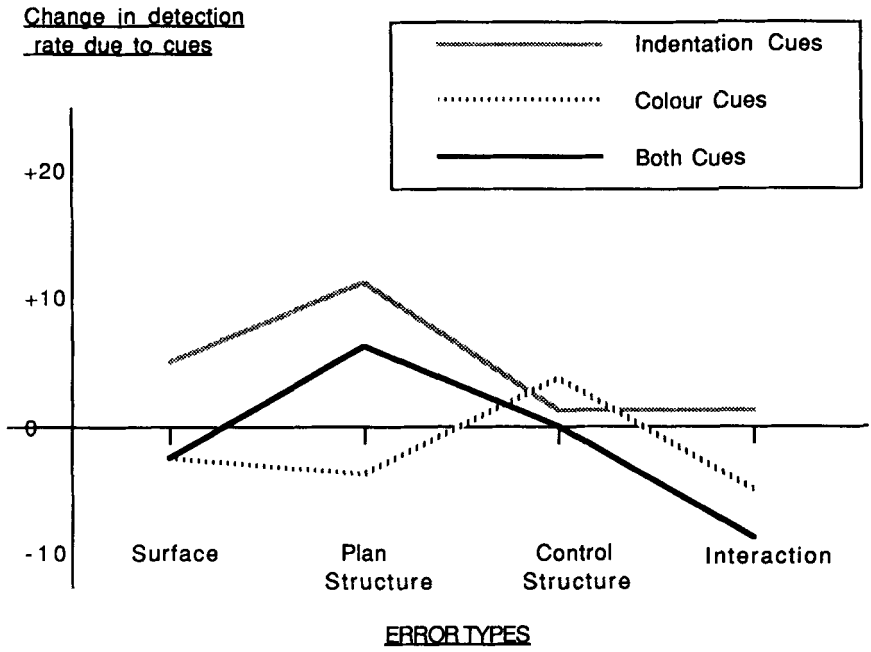

FIG. 4 Change in the error detection rates for the Basic programs caused by the presence of cues.

Because of the lack of any effects for the BASIC programmers and because many of the BASIC programmers commented that there was a large proportion of control-flow bugs, it was felt necessary to examine the error descriptions made by the two groups of programmers. Figure 5 shows for each bug type the percentage of those detected that were described in terms of control flow. It is clear that this proportion is considerably higher for the BASIC programs than for the Pascal programs, suggesting that the BASIC programmers may use control structure as their predominant view of the program.

This tendency suggests that indentation should be the most useful cue to the BASIC programmers, as it provides cues to the information structure, which they find most useful. Re-examination of Figure 4 reveals that in three of the four error types the best bug detection rate occurred with the indented programs, though this difference did not approach significance.

Thus, in summary, all of the hypotheses made in the introduction have been supported by the data from the Pascal programmers, whilst none of them has been supported by the data from the BASIC programmers.

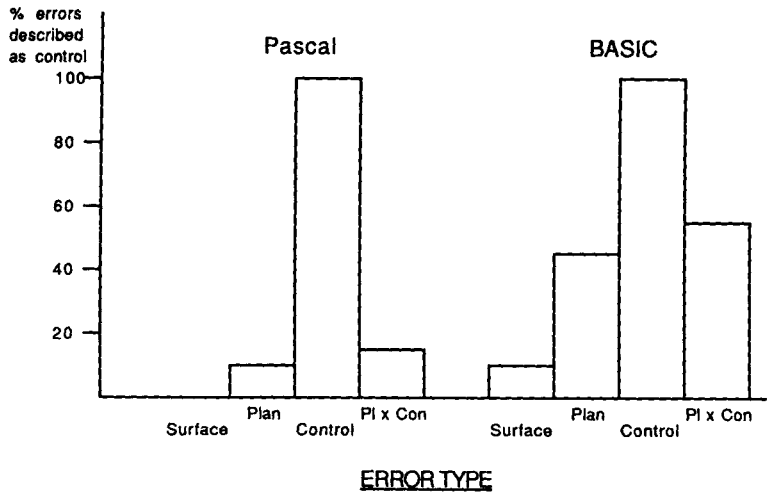

FIG. 5 Percentage of errors detected that are described in terms of control-flow.

Discussion

In relation to the three questions posed in the Introduction, the results clearly show that

1. plan structures are psychologically meaningful to Pascal programmers, because perceptual cues to them bring about a significant improvement in performance;
2. as plan structure cues do not improve performance on non-plan-related tasks, plans do not represent the deep structure of the problem but are a non-syntactic view of the code, with which experts are proficient;
3. these Pascal plans do not generalise to BASIC, suggesting that expert BASIC programmers do not use the same view of the program as Pascal experts, possibly being more influenced by control-flow.

Fortunately these answers are coherent, in that it would make sense for plans to be a mental language of programming and yet not to generalize to other languages. Likewise, if plans are only an alternative information structure within the program, then we should expect them to be partly determined by the language being used.

The first two results do not lessen the importance of the plan concept, but they suggest that tools and languages should not be designed solely around plans. The emphasis needs to be on tools and languages that facilitate the interleaving of different plan components when writing, and the unravelling

of components when reading. Furthermore, the effectiveness of cues to plan structures reveals the importance of a clearer definition of plans, as from a formal definition it would be possible to provide such cues automatically.

In furtherance of this end, we should turn our attention to the third result—the failure of the results to generalize across languages—because this means that formalizing the Pascal plans, rather than a more general concept, may be inadequate for the provision of tools for other languages.

But before discussing these matters in too great a detail, it is essential to consider the possibility that these results arise simply because of uninteresting population differences, as there is clearly a confounding of subject and language effects. An obvious argument would be that the BASIC programmers were not as experienced, but although we have no data on their amount of programming experience, they were pursuing a course in Computational Mathematics which required a considerable quantity of programming. It is unlikely, therefore, that there was any sizeable difference in the programming experience of the two groups.

A second possibility, which is more plausible, is that there were major differences in teaching strategies between the two groups. However, the main question of this paper is whether expertise in programming *necessarily* involves the use of plan structures when writing and reading programs. If the acquisition of plan structures is the defining quality of expertise, then differences in teaching stategy etc. should not affect the nature of expertise. For example, it is reasonable to argue that the nature of expertise in chess does not depend on teaching strategies, because the nature of attack and defence and the configurations that represent them are inherent in the game. Similarly, programming plans are assumed to be inherent in the problem, not in the language or the teaching, and the development of such schemata/plans is dependent upon experience, not on teaching. Our data shows that this is an indefensible viewpoint for programming expertise and that different forms of expert knowledge arise as a result of either language or teaching differences. Given that we have little information about the teaching strategies, but we do know plenty about the language differences, the remainder of this discussion will consider what influence they may have. It is hoped that by so doing some of the important teaching factors might suggest themselves. This can be best achieved by considering more carefully what programming plans might be.

A Clarification of Plans

Could Plans have Generalised Across Languages?

If we consider Rist's (1985) description of plan structures, then it is apparent that the original question of whether plans would generalize to BASIC is meaningless. In Rist's description a plan is a mapping between

some aspect of the problem to be solved and lines of program code. Clearly this mapping could not be exactly the same in different languages, because the code itself differs. Thus, there is a need to clarify exactly what was intended by the question, and the implications of this for the results.

There are in fact three components to Rist's plan structures: first there is the knowledge of the parts of the problem to be programmed, then there is the knowledge of the programming language, and finally there is the mapping. Rist's view of the relationship between these components is that they together form a plan structure as acquired by an expert programmer in an all-or-none manner.

An alternative, however, is to view the first two as knowledge to be acquired and the third component as an automated skill (in the sense of Underwood, 1982). In this way a programmer can possess either of the first two components independently, but the third depends upon the existence of the other two. In this view a programmer may fail to display knowledge of plans for a variety of reasons.

1. A programmer may lack the knowledge with which to analyse the problem into its parts; for example, a progammer may be unfamiliar with the task, or components of the task, being solved (e.g. calculating an ANOVA) and therefore be unable to take advantage of this.
2. A programmer may lack the knowledge of how to code a particular part of the problem, even when the required algorithm is known. Such knowledge is likely to be related to experience. This possibility is supported by the fact that in Rist (1986) there was evidence that novices could appreciate the plans used in a program.
3. Finally, a programmer may possess both these pieces of knowledge, but the process of mapping one onto the other may not be known, or automated. This automatization process will also be related to programming experience.

Given this view of plan structures, the original question about the generalization of plans can be understood. The question addressed by these experiments was whether the mapping between *equivalent* pieces of code and their role in the problem was *similarly* automated in both groups of programmers. The results suggest not, and it is unlikely that this is due to differences in the knowledge aspect of plans, as there is no obvious reason for the two groups to differ in this—the language-independent knowledge can be assumed equal in programmers of such similar backgrounds, and the language-dependent knowledge is unlikely to differ, as both languages require similar algorithms to solve these problems.

The question that remains, therefore, is why should the mapping become automated for Pascal programmers, but not (or less so) for BASIC programmers? Two reasons suggest themselves:

1. BASIC does not contain stereotypic fragments;
2. BASIC programmers fail to detect these fragments.

In fact, both of these may be true. Firstly, the greater variety of syntactic constructs and the structure within Pascal (compared with the BASIC used in this experiment) provides the programmer with an easier task, as there is a greater chance that stereotypes will occur. Second, factors such as the declaration and initialization of variables in Pascal provides an extra reference point for each variable, suggesting its role before the main processing loop. Also, the fact that Pascal programmers have a compiler than can detect most surface errors leaves them with more time to study the higher-level structures of the program. This argument is supported by the fact that it was only in the surface errors that there was a consistent difference (across all cue conditions) in the error detection rate between the two languages (see Appendix 1 and Table 1).

The general point underlying these explanations is that the Pascal programs are more discriminable from each other (cf. Fitter & Green, 1981), allowing the programmer to infer the role of a particular statement more easily. This is a property that we have termed *role-expressiveness* (Gilmore, 1986b). The argument is that role-expressiveness is a property of the language that enables the automatization of the mapping between problem and programming knowledge. In general, languages that promote role-expressiveness will be easier to use.

Role-Expressiveness

There are three components to role-expressiveness:

1. *Discriminability:* This is simply whether the notation provides easy access to chunks of code, where a chunk may be an individual statement, or a group of statements. The first step in perceiving the role of some statement is being able to discriminate that line from those that surround it.

2. *Statement-Structure Mapping*: The structural role of a statement is its role within the program itself, independent of the problem being solved. Unstructured languages (such as some BASICs) may use the same piece of code for more than one purpose (e.g. as an initialization and as an update). Other languages make it harder to see what action is being performed by some statement. For example, Prolog does not distinguish between instantiated and uninstantiated variables—the Prolog command (SUM X Y Z) may be adding X to Y, subtracting either of them from Z, or validating that a sum is correct.

3. *Statement-Task Mapping.* Programmers need to understand not only the structural role of program statements, but also their task role. Difficulties arise here in languages that force the programmer to include statements with

```
10 add_egg
20  stir
30 IF need_more_egg GOTO 10
40 IF ready_to_eat GOTO 70
50  add_oil
60 GOTO 20
70  use_mayonnaise
```

FIG. 6 An unstructured program for making mayonnaise. Line 10 has two roles—adding the initial egg and adding extra egg later (after Green, 1980).

either many task roles or no task role. Figure 6 provides an example of multiple task roles, in which the opening statement of the program has two task roles. A different example of difficulty at this level can be seen in Soloway, Benar, and Ehrlich's (1983) study of novices' looping errors in Pascal, in which they observed that novices who tried to force the problem into a "read/process" loop made errors when they were required to test the terminating condition twice. The cause of this seems to be that the second test has no clear task role. Informally we have noted a similar effect with novice POP11 programmers learning tail-recursion. They struggle when the syntactic structure of their program requires that the recursive function be called twice. An example of such a program is given in Figure 7.

This concept of role-expressiveness is very important in the automation of the mapping between program code and its function, but it is not the only determinant of programming success (see Gilmore, 1986b, and Green, Bellamy, & Gilmore, in preparation, for others). The difficulty that Lisp novices have understanding the CONS function (Anderson, Farrell, & Sauers, 1984) can be interpreted as a problem at the statement-structure mapping. The automation of this mapping may give rise to a different type of plan knowledge. Different teaching strategies may provide environments in which different emphases are placed on these different aspects of program-

```
define countx(list, x);
if list = [] then
    0
elseif hd(list) = x then
        countx(tl(list),x) + 1
else
        countx(tl(list),x)
endif
enddefine;
```

FIG. 7 A POP11 program whose syntactic structure requires the duplication of the tail-recursion. The problem can be avoided by the use of a variable.

ming knowledge, thus leading to differences in the nature of expertise. Much more research is needed on the importance of language and educational factors in determining the development of expertise.

SUMMARY

This paper has presented data that reveal a need to clarify the concept of programming plans and their role in expertise. We claim that plans do not represent the underlying deep structure of programming problems, and that the content of plans as observed in Pascal experts is not the only possible content in plan knowledge. Plans are described as having three components:

1. "naive" knowledge of methods of solving the problem of interest; this may be affected by knowledge of the syntax of the programming language;
2. knowledge of how to achieve particular effects within the programming language;
3. an "automatic" process that maps (1) onto (2).

This model suggests that plans may be of two types: Firstly, the mapping may be triggered by (1), which includes most of the plans decribed by the Yale project; secondly, the mapping may be triggered by (2)—for example, a mapping may exist between the use of lists to solve some problem and the Pascal representation of list structures. The important factor is that expertise can develop at all three levels independently. This has implications for development of programming tools and teaching aids.

Although our knowledge of the formation of automatic cognitive processes is not sufficient to allow us to help students to construct the mapping, it is reasonable to suppose that increasing a student's awareness of the two types of knowledge and providing practice in mapping one onto the other should be of great benefit. This explains, in part, the success of the structured programming school, because their emphasis on problem decomposition should allow students to perceive the common methods of solution within problems. Likewise, tools such as Bridge (Bonar & Cunningham, in press) and Proust (Johnson, 1986) provide Pascal students with considerable exposure to knowledge of Type (2). However, whereas structured programming and knowledge of type (1) are language-independent, tools such as Bridge and Proust are not. Thus, the important next steps are to develop methods of predicting what knowledge of Type (2) would be for other languages and to test these predictions empirically. If the results support this model of expertise, then tools for languages other than Pascal may be possible.

In conclusion, the concept of a programming plan is of central importance

to the development of powerful programming environments and usable programming languages, but determining their nature in language-independent terms is an essential priority, which could have a significant impact on general theories of the nature of expertise.

REFERENCES

Anderson, J. R. (1985). *Cognitive psychology and its implications*. New York: W. H. Freeman.

Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in Lisp. *Cognitive Science, 8*, 87–129.

Bower, G. H., Black, J. B., & Turner, T. J. (1979). Scripts in text comprehension and memory. *Cognitive Psychology, 11*, 177–220.

Bonar, J. & Cunningham, R. (in press). Bridge: An intelligent tutor for thinking about programming. In J. Self (Ed.), *Intelligent computer-aided instruction*. London: Chapman & Hall.

Fitter, M. E. & Green, T. R. G. (1981). The art of notation. In M. Coombs & J. Alty (Eds.), *Computing skills and the user interface*. London: Academic Press.

Gilmore, D. J. (1986a). Structural visibility and program comprehension. In M. D. Harrison & A. F. Monk (Eds.), *People and computers: Designing for usability*. Cambridge: Cambridge University Press.

Gilmore, D. J. (1986b). *The perceptual cueing of the structure of computer programs*. Unpublished PhD Thesis, Psychology Dept., University of Sheffield, England.

Gilmore, D. J. & Green, T. R. G. (1984). Comprehension and recall of miniature programs. *International Journal of Man–Machine Studies, 21*, 31–48.

Green, T. R. G. (1980). Programming as a cognitive activity. In H. T. Smith and T. R. G. Green (Eds.), *Human interaction with computers*. London: Academic Press.

Green, T. R. G., Bellamy, R. K. E., & Gilmore, D. J. (in preparation). Psychological dimensions of interaction languages.

Johnson, W. L. (1986). *Intention-based diagnosis of errors in novice programs*. Report No. 395, Dept. of Computer Science, Yale University, New Haven, Connecticut.

Johnson, W. L., Soloway, E., Cutler, B., & Draper, S. (1983). *Bug catalog 1*. Research Report No. 286, Computer Science Dept., Yale University, New Haven, Connecticut.

Kesler, T. E., Uram, R. B., Magarah-Abed, F., Fritsche, A., Amport, C., & Dunsmore, H. E. (1984). The effect of indentation on program comprehension. *International Journal of Man-Machine Studies, 21*, 415–428.

Miara, R. J., Musselman, J. A., Navarro, J. A. & Shneiderman, B. (1983). Program indentation and comprehensibility. *Communications of the ACM, 26*, 861–867.

Reason, J. (1984). Lapses of attention in everyday life. In R. Parasuraman and D. R. Davies (Eds.), *Varieties of attention*. London: Academic Press.

Rist, R. (1985). *Program plans and the development of expertise*. Unpublished manuscript, Dept. of Computer Science, Yale University, New Haven, Connecticut.

Rist, R. (1986). Plans in programming: Definition, demonstration and development. In E. Soloway and S. Iyengar (Eds.), *Empirical studies of programmers*. Norwood, N. J.: Ablex.

Schank, R. C. & Abelson, R. P. (1977). *Scripts, plans, goals and understanding*. Hillsdale, N.J.: Lawrence Erlbaum Associates, Inc.

Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM, 26*, 853–860.

Soloway, E. & Ehrlich, K. (1983). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering, 10*, 565–609.

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1982). What do novices know about

programming? In A. Badre and B. Shneiderman (Eds.), *Directions in human-computer interaction*. Norwood, N.J.: Ablex.

Spohrer, J. C., Soloway, E., & Pope, E. (1985). A goal-plan analysis of buggy Pascal programs. *Human-Computer Interaction, 1*, 163–207.

Underwood, G. (1982). Attention and awareness in cognitive and motor skills. In G. Underwood (Ed.), *Aspects of consciousness, Vol. 3*. London: Academic Press.

Revised manuscript received 23 July 1987

APPENDIX 1

The *F* values for the simple effects of language

	Error Types			
	Surface	*Plan Structure*	*Control Structure*	*Interaction*
No cues	**27.6**	0.2	3.7	0.4
Indentation	**27.2**	**5.7**	*0.4*	1.6
Colour	**7.2**	**5.0**	**8.9**	0.16
Both	**5.0**	*0.32*	*1.6*	**4.9**

$df = 1,168$. $F_{crit} = 3.92$.

Bold values indicate significant differences.

Italic values indicate those where Pascal programmers detected more errors.