The Psychological Study of Programming

. A. SHEIL

Cognitive and Instructional Sciences, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94305

Most innovations in programming languages and methodology are motivated by a belief that they will improve the performance of the programmers who use them. Although such claims are usually advanced informally, there is a growing body of research which attempts to verify them by controlled observation of programmers' behavior. Surprisingly, these studies have found few clear effects of changes in either programming notation or practice. Less surprisingly, the computing community has paid relatively little attention to these results. This paper reviews the psychological research on programming and argues that its ineffectiveness is the result of both unsophisticated experimental technique and a shallow view of the nature of programming skill.

Keywords and Phrases: programming, cognitive psychology, programming languages, programming methodology, human factors, software engineering

CR Categories: 4.0, 4.29

INTRODUCTION

As practiced by computer science, the study of programming is an unholy mixture of mathematics (e.g., DIJK76), literary criticism (e.g., KERN74), and folklore (e.g., BROO75). However, despite the stylistic variation, the claims that are made are all basically psychological; that is, that programming done in such and such a manner will be easier, faster, less prone to error, or whatever. How compelling we find such formal, stylistic, and anecdotal arguments reflects primarily the degree to which they appeal to our own "common sense" model of the cognitive processes involved in programming. However great their appeal, though, the methodological recommendations of computer science should be recognized as *empirically testable, psychological hypotheses*. Unlike mathematics, literary criticism, or folklore, a discipline of computer science has an obligation to validate these claims.

The results of psychological experiments have, indeed, often been used as ammunition in disputes over computing practice. One of the earliest contributions was Sackman's study of time sharing [SACK70], conducted in response to the then current debate on the relative efficacy of batch and time-shared processing. Weinberg's well-known book, *The Psychology of Computer Programming* [WEIN71], in addition to making a forceful claim for the importance of psychological considerations in computing, accommodates the then current debate on programmer team organization by treating the sociology of programming groups at length. More recently, much of the work on programming language constructs and practices discussed both in this review and in Shneiderman's book, *Software Psychology* [SHNE80], is clearly motivated by the structured programming movement.

Sadly, however, psychological data have been at best a minor factor in these debates.

CONTENTS

Sackman's inability to find decisive advantages for time sharing did little to slow its spread. Although one could certainly find points to object to in his studies, the unfortunate fact is that they were largely ignored by a discipline that did not want to hear about them. Despite the greater acceptance now than then of the relevance of experimental results, little of the work on the psychology of programming shows any sign of challenging computer science's established wisdom on programming and programming methodology. Much of what follows is an attempt to explore why this is so.

1. PSYCHOLOGICAL THEORY AND BEHAVIORAL EVALUATION

Before considering the behavioral evaluations of programming technology, one might ask whether the relevant insights could be obtained from conventional psychological *theory*. An analysis based on a psychological theory which has been validated in a different domain would have the considerable authority of independent grounding, as opposed to the somewhat ad hoc flavor of technology-driven experimentation. Unfortunately, although some psychological theory is very suggestive, it usually lacks the robustness and precision required to yield exact predictions for behavior as complex as programming. As a result, the psychological work on programming consists mainly of atheoretical evaluations motivated directly by the concerns of contemporary computing practice.

Such evaluations might, at first sight, seem to be a straightforward matter of comparing the performance of groups of similar programmers using different programming techniques. Unfortunately, the complexity of programming behavior makes the execution and interpretation of such comparisons anything but straightforward. A rudimentary appreciation of these difficulties is necessary to motivate the methods used, lest they seem pointlessly abstruse.

The empirical methods span a spectrum defined by the tension between the conflicting goals of *reliability* and *generalizability*. Reliability (or *control*) is the degree to which observed relationships are systematic rather than circumstantial, whereas generalizability is the degree to which such relationships occur or are significant in situations other than the one observed. Observations made in real-world situations are at one extreme, in that any findings are known to apply to at least *one* real-world situation. Their generalizability to others, however, is clouded by the fact that real-world programming performance varies in too many uncontrolled ways to allow reliable interpretation of its causes. Intervention to control these extraneous influences (e.g., by reassigning programmers within working groups to equalize the level of ability across the groups) is apt to disrupt the organization, with totally unpredictable effects on the phenomenon being observed. Indeed, the very act of introducing some technical innovation for the purposes of evaluating it may cause changes of behavior that are unrelated to any properties of the innovation, a phenomenon known as the *Hawthorne effect.* These difficulties are common to most *in situ* investigations of complex behavior, and a variety of specialized techniques (for a discussion of which see, e.g., CAMP79) are required to circumvent them.

The alternative to real-world experimentation is to construct artificial experimental situations in which extraneous influences can be either eliminated or controlled. The construction of realistic experimental programming situations that differ in only (a few) controlled ways is, however, both difficult and subject to its own forms of bias. Creating two or more similar programming environments by altering an existing environment can cause interference effects among programmers familiar with the status quo. Completely novel environments, on the other hand, whether specially constructed for the experiment or obtained by studying programmers with no experience of some existing environment, require that the programmers being studied be thoroughly trained for their new environment, lest learning transients dominate the results. Similar issues arise with respect to the type and size of the programming task used. Failures of either realism or control can result in any differences between technologies being concealed by extraneous variation.

The difficulty of reliable experimentation using complete programming tasks suggests experiments which focus on either isolated aspects of the programming task or the psychological claims that implicitly underlie different programming techniques. Thus one might evaluate the suitability of using transfer of control to express conditional action by investigating how well people can formulate and understand simple procedures in various stylized natural languages, such as those commonly used to express sets of instructions such as recipes. Differences in behavior could be used to support claims about the appropriateness of certain uses of transfer of control within programming languages. The attractiveness of this approach is that its distance from real-world programming permits much easier and more tightly controlled experimentation. That same distance, however, makes the argument from experimental findings to normal programming practice very perilous. Not only must it be shown that the experimental situation taps psychological processes which occur during programming, but those processes must also be shown to account for a significant proportion of programming effort. Neither of these demonstrations, needless to say, is at all straightforward.

2. SCOPE OF THIS REVIEW

The behavioral research on programming can be roughly separated into studies of

- *programming notation,* which includes the effects of programming language features such as control structures;
- *programming practices,* such as programming methodologies, tools, and aids, such as the use of comments, indentation, and flowcharting;
- *programming tasks* or behavior specific to certain aspects of programming, such as learning a programming language, coding, and debugging;
- *programming management,* which includes issues relating to the management, review, and organization of programming groups.

In addition, a variety of other studies have investigated such factors as the programmer's physical environment, personality, and motivation, and the predictive value of various measures of program complexity. While all of these are potentially significant, this review concentrates exclusively on studies in the first three areas, which form a coherent body of research on the *cognitive* processes underlying programming. The other studies cover a very different and very diverse range of topics, such as the social context in which the programming takes place.

Even within the area of cognitive psychology this review is not exhaustive. In addition to the usual caveats of unintentional oversight and limited space, a detailed enumeration of work in this area has recently been published [SNHE80]. The intent here is to explore its overall direction, current state, and future prospects.

3. STUDIES OF PROGRAMMING NOTATION

As one might expect, given the diversity of programming languages and the intensity of feeling about them in the computing community, the study of programming notations is both the largest and the most controversial area. As one might further expect in light of the structured programming debate, choice of control constructs dominates as subject matter.

3.1 Conditionals

The use of GOTOs to express conditional and loop structures was one of the first targets of the structured programming critique [DIJK68], so it is not surprising that notation for conditionals was one of the first subjects of psychological study. An early study by Sime, Green, and Guest [SIME73] compared a nested IF-THEN-ELSE notation with one that used explicit transfers of control. In this experiment nonprogrammers were asked to compose several sets of instructions for a simple mechanical device using one of the two notations. Each set of instructions was to select the appropriate action (a method of cooking) to be applied to an input object (a vegetable) on the basis of its properties (color, texture, etc.). For each problem the subjects were presented with a specification in the form of a set of action-attributes pairs, and measures were taken of both the amount of time and number of errors required to produce a correct program. On both counts the IF-THEN-ELSE notation was found to be superior to test-and-jump. They later [SIME77] replicated this result in an experiment which included a third notation wherein the predicate was repeated in each clause of the conditional (i.e., IF *pred* *action*$_1$ NOT *pred* *action*$_2$ END *pred*, as shown in Figure 1). Their results presented in Table 1 indicate that, for the simple (one-word) predicates they used, the repeated predicate form is slightly superior to the IF-THEN-ELSE form, although their version of the latter is a little prolix in its use of BEGIN . . . END pairs.

Similar results were later obtained by Green [GREE77] for two comprehension tasks using the same notations. Experienced professional programmers were presented with programs written in one of the three different languages and asked to specify either the action that would be carried out for an input with given properties or what those properties would have to have been given that the program performed some given action. An example of the forward reasoning task, using the programs in Figure 1, would be to specify the action to be taken (grill) if the input were hard and not green; the backward reasoning task would be to specify what properties the input must have had given that it was grilled. The amount of time taken to answer these questions for each of the three languages is shown in Table 2. Both types of questions were answered faster for nested language programs than for jump programs. In addition, a smaller difference was found between the two nested languages for reasoning backward from effect to preconditions.

TABLE 1. ERROR DATA FOR PROGRAMMING TASK[a,b]

	Language			
	Jump	*Nested*	*Repeated*	*p-level*
Semantic errors	0.40	0.07	0.04	.01
Syntactic errors	0.20	0.85	0.13	.05
Error lifetimes	1.06	1.60	0.09	.02

[a] From SIME77.
[b] The first two lines give the number of the indicated type of error per problem. The last line gives the number of additional attempts after an initial error. All entries are medians.

These studies make a reasonable case for the superiority of nested conditional structures over those using jumps. The authors explain this difference in terms of redundant perceptual encoding (specifically, that the indentation of nested programs provides a secondary clue to their logical structure). This seems reasonable, although it is puzzling that no direct test of this explanation (e.g., by comparing jump and nest programs, both unformatted) has ever been reported.

By contrast, the explanation of the difference between the repeated predicate and the IF-THEN-ELSE notations in terms of the latter requiring the reader to negate predicates when reasoning backward from

```
     IF hard GOTO L1
     IF tall GOTO L2
     IF juicy GOTO L3
     roast STOP
L1   IF green GOTO L4
     peel grill STOP
L2   chop fry STOP
L3   boil STOP
L4   peel roast STOP
```

Test and jump

```
IF hard THEN
BEGIN peel
  IF green THEN
  BEGIN roast
  END
  ELSE
  BEGIN grill
  END
END
ELSE
BEGIN
  IF tall THEN
  BEGIN chop fry
  END
  ELSE
  BEGIN
    IF juicy THEN
    BEGIN boil
    END
    ELSE BEGIN roast
    END
  END
END
```

Nested

```
IF hard peel
  IF green roast
  NOT green grill
  END green
NOT hard
  IF tall chop fry
  NOT tall
    IF juicy boil
    NOT juicy roast
    END juicy
  END tall
END hard
```

Repeated conditions

FIGURE 1. Three notations for conditionals (from SIME77).

ELSE alternatives is much less convincing. The use of secondary visual clues to explain the superiority of nested structures suggests that the efffectiveness of these notations will be very sensitive to differences in formatting conventions. Nested languages, however, permit many variations both of indentation and choice of delimiters, and this factor is neither systematically varied nor controlled in these studies. Specifically, the superiority of the repeated predicate conditional over the standard IF-THEN-ELSE form might simply be an effect of the formatting used for the latter. Simply glancing at Figure 1 shows that the two nested languages are not matched in terms of number of symbols, the amount of space they occupy, or the relative spatial displacement of related clauses. Intuitively, reading programs in the IF-THEN-ELSE language seems to require a significant amount of effort to climb over the forest of redundant BEGIN . . . END brackets. Neither can it be claimed that these uncontrolled differences are inherent in the two types of nesting, as a variety of more concise techniques has been used in various programming languages to delimit the clauses of IF-THEN-ELSE constructions. Without experiments that control for these alternative sources of variation, there is simply no evidence for the claim that the difference between the two nested languages is due to the mental overhead of the implicit negation required for ELSE clauses. The differences between the two nested languages in terms of the

TABLE 2. CONDENSED REACTION TIME DATA ON TWO COMPREHENSION TASKS[a,b]

Language	*Session* 1	2	3	*Mean*	*p Level*
	Forward Reasoning Task				
Jump	7.415	6.050	5.135	6.200	.032
Nested	7.095	5.670	4.760	5.842	
Repeated	7.470	5.870	4.860	6.067	ns
Mean	7.327	5.863	4.918		
	Backward Reasoning Task				
Jump	10.090	7.590	6.510	8.063	<.001
Nested	9.345	7.040	6.120	7.502	
Repeated	8.825	6.605	5.675	7.037	.011
Mean	9.420	7.078	6.085		

[a] From GREE77.
[b] One factor (number of conditionals) has been collapsed out. Times are in seconds. Probability levels contrast Jump with both nested forms, and Repeated with Nested.

number and positioning of symbols provide more than sufficient reason to expect differences in performance.

GREE77 also provides an example of another very striking effect which is often found in behavioral studies of programming. Each participant in this experiment attended three sessions held on consecutive days. The reaction time data presented in Table 2 show large practice effects (improvement in performance across sessions) for all three languages in both experiments. Whereas the effect of the different languages is to change the mean reaction time by amounts which range from 4 to 15 percent, the effect of a single session of practice ranges from 13 to 27 percent! Even though these practice effects do not interact with those due to language, their size relative to the effects of notation provides one measure of the importance of those notational differences.

Practice and experience effects like this have been noted in other studies, often interacting with or extinguishing the linguistic effects being studied. Sime et al. [SIME73] reported that with practice the differences between the two languages they studied became nonsignificant. A study of the FORTRAN arithmetic and logical IF statements [SHNE76] found the logical IF to be significantly easier for novices but found no such differences for more experienced programmers. These interactions between notation and experience suggest that the effects of notation being measured in these experiments may be somewhat evanescent.

3.2 Control Flow

The evidence for the structured programming position on control structures other than the conditional is considerably more mixed, possibly because of the difficulty of developing experimental materials that differ only in the degree to which they are structured. Weissman [WEIS74] used two PL/I programs, each written at three levels of structuring, and found that the structured version led to greater confidence on the programmer's part, but no reliable differences in performance on either comprehension measures or debugging performance, although most of these measures were tending to favor the structured version. He speculated that the effects would have become clearer with larger programs than those he used (50 and 100 lines, respectively) but points out that such programs would be difficult to use in the necessarily restricted time of an experiment.

TABLE 3. RESULTS FROM STRUCTURED PROGRAMMING STUDY[a,b]

	Composition			*Modification*		
	Structured		*Unstructured*	*Structured*		*Unstructured*
Test runs	14.63		8.87	10.06	=	10.25
Time taken	14.75	=	11.69	9.13		13.06
Easy to write	4.00		5.19	5.13	=	4.63

[a] From LUCA76.
[b] Time units were not reported. "Easy to write" is a subjective evaluation with higher scores indicating greater ease. Differences are significant at .05 level unless marked by =.

Lucas and Kaplan [LUCA76] contrasted performance on composing one program and modifying another using GOTO-less and standard PL/I. On their performance and attitudinal measures, the only reliable differences between the two groups were that writing GOTO-less programs required more test runs (and, perhaps not incidentally, was thought to be less easy) but no more programming time, whereas modifying the GOTO-less programs took less programming time but no fewer test runs (see Table 3). This asymmetry may reflect the fact that no explicit training in GOTO-less programming was provided, and its sudden imposition as a constraint may have disrupted the participants' established programming styles in the composition task.

Sheppard et al. [SHEP79] studied the performance of professional programmers on small (~50 statements) FORTRAN programs that were structured in a number of different ways, including chaotically. The latter were significantly more difficult both to memorize and modify, but no differences were found, either for these tasks or for debugging, between "naturally" structured programs and versions that were more strictly structured, flow-graph reducible, or written in FORTRAN77 (a dialect with structured programming constructs such as IF . . . THEN . . . ELSE . . .).

All told, this is fairly poor support for computer science's dominant programming paradigm. Shneiderman's review is more sanguine, stating "These controlled experiments and a variety of informal field studies indicate that the choice of control structure does make a significant difference in programmer performance. Evidence supports the anecdote that the number of bugs in a program is proportional to the square of the number of GOTOs..." [SHNE80, p. 81]. Unless the (uncited) "informal studies" which are alluded to are very compelling, the evidence suggests only that deliberately chaotic control structure degrades performance. These experiments provide virtually no evidence for the beneficial effect of any specific method of structuring control flow.

On the other hand, one could argue that the results of these experiments simply do not bear on the methodology of structured programming that they purport to test. As Dijkstra points out in the title of his book, *A Discipline of Programming* [DIJK76], any reasonable programming methodology is a *discipline,* a way of thinking, not just a collection of programming constructs. The empirical claim of structured programming is that a programmer who approaches problems in a certain way will be more effective. The syntactic constructs are appropriate to that approach, but they are not themselves that approach. Therefore there is no reason to believe that their presence or absence will, by itself, have any significant impact. Either the programmer understands the structured approach to programming, in which case her code will reflect it (whether or not structured control constructs are available), or the programmer does not, in which case the presence of syntactic constructs is irrelevant.

The experimental manipulations of LUCA76 are quite inappropriate from this point of view. Simply outlawing the use of a language feature without providing any motivation or alternative strategies may replicate some of the foolishness of the early days of the structured programming movement but is hardly likely to produce much else other than resentment. The same assumption, that the value of a program's being well structured is independent of the attributes of the person reading it, also underlies the investigation of the tractability of differently structured programs in SHEP79. Since their programmers were not trained in any particular programming methodology, their finding that strictly structured programs were no easier to deal with than "naturally" structured ones may reflect only that such "natural" structuring was most familiar to their participants. Likewise, the result that FORTRAN77 programs were no better understood than "naturally" structured ones is not at all surprising given that less than 10 percent of their participants had had any exposure to FORTRAN77.

3.3 Data Types

One of the major distinctions among programming languages is whether the objects in the language are required to be known to be of a certain *type* (e.g., integer, string), either *statically* (at compile time, as in PASCAL), *dynamically* (at run time, as in LISP), or not at all (as in BCPL). This is a particularly critical issue for system programming languages, where both reliability and expressive freedom are highly desirable.

Gannon conducted an experiment designed to compare the error proneness of statically typed and typeless languages [GANN77]. His method was to provide both statically typed (with integer and string types) and typeless (e.g., arbitrary subscripting of memory) extensions to a common language core, thus generating two languages which were essentially matched except for differences in the type conventions. A class of 38 students programmed the same problem in both languages, with half doing it in each order. Gannon's careful analysis makes a strong case for the superiority of the statically typed language in controlling errors.

Unfortunately for the argument for static typing, Gannon's analysis of the source of the errors makes it clear that few of the errors made in the typeless language had to do with the lack of static checking, but mostly reflected problems with data representation. This, as Gannon points out, is an inherent confound, as a statically typed language must provide a basic set of operations on the data types it supports. Although Gannon took pains to minimize the extent of the built-in semantics, his error analysis clearly shows the bulk of errors made in the typeless language to be in the code that provided the operations on strings that were built into the statically typed language. This certainly affords evidence for extending the facilities offered by programming languages, but little evidence for the value of static typing.

Gannon's study also provides another instance of the experience effects noted in the studies of conditionals, as the differences between the two languages were much reduced for both more competent (as measured by course grades) and more experienced (as measured by number of programming languages known) participants.

3.4 Everything at Once

Perhaps the most ambitious attempt to contrast programming language designs empirically is Gannon's thesis [GANN75, GANN76]. Flush with the confidence of the structured programming movement at its height, Gannon and Horning took an existing language that was in use as a teaching language at the University of Toronto and made nine separate modifications based on their analysis of its deficiencies. The resulting language was constrasted with its predecessor in a two-group (stratified by ability) experiment using student programmers, who were asked to complete two moderate size (75–200 line) programming problems in their assigned language. The evaluation measure was the error rate for those participants who completed both problems (an astonishingly low 49 percent of the original group).

Unfortunately, the overall error rates of the two languages were not significantly different, leaving Gannon the unrewarding task of tracing the source of each of the 3937 errors (!!) in an attempt to construct causal interpretations by *post hoc* analysis. The language features found to be more prone to error (as measured by either error counts, occurrences, or persistence) included untraditional operator precedence, assignment being an operator rather than a statement, semicolon being a separator rather than a statement terminator, use of bracketing to close both compound statements and expressions, and the inability to use named constants.

The problem with this analysis is that it is very unclear how to interpret the results. For example, Gannon and Horning argue that the persistence of assignment errors in the original language "calls into serious question the treatment of the assignment symbol := as 'just another operator'" [GANN75, p. 19]. These results, however, are open to far too many plausible alternative explanations. One such alternative is that the errors may result from confusion due to the use of an assignment symbol which closely resembles another valid operator (=) which is itself the assignment operator in other well-known languages, one of which (PL/I) is mentioned as being familiar to some of the students. This, possibly compounded by the use of nonstandard precedence, may well be a specifically deadly combination of language features. Gannon and Horning claim that use of a different symbol (such as ←) might "probably avoid some of these errors, but provide no better error detection." Then again, it might avoid all of them. One just cannot tell on the basis of a single observation.

3.5 Summary

Our discussion of the effects of various programming notations has raised two significant general issues. First, given the small sizes of and inconsistencies among the reported effects, it is not even clear that notation is a major factor in the difficulty of programming. The study of programming languages has been central to computer science for so long that it comes as a shock to realize how little empirical evidence there is for their importance. Second, many of these effects tend to disappear with practice or experience. This raises some doubt as to whether these results reflect stable differences between notations or merely learning effects and other transients that would not be significant factors in actual programming performance.

4. STUDIES OF PROGRAMMING PRACTICES

Programming "practices" include such things as flowcharting, prettyprinting, variable naming, and commenting. The empirical investigation of their efficacy has been prompted both by their frequent recommendation by textbooks on programming (e.g., KERN74) and by their apparent accessibility to simple experimental test. A claim that some such practice as extensive use of comments will significantly improve some measure of programming effectiveness is easily tested, as the practice can be imposed in any appropriate existing programming environment. The study of notations, by contrast, not only requires the construction of quite elaborate experimental environments, but it is much more difficult to vary aspects of the notation independently, as such changes tend to interact with other aspects of the environment.

4.1 Flowcharting

Shneiderman et al. carried out a series of experiments designed to determine which kinds of programming tasks were most enhanced by use of detailed flowcharting [SHNE77b]. Five successive experiments, with different tasks and measures, failed to reveal any reliable advantage of flowchart use. As Brooks pointed out [BROO80], one can explain the lack of results in two of these studies as "ceiling" effects (i.e., the scores of both experimental groups were so close to the maximum that there was no room for differences to occur). One could also explain away the lack of results in the remaining studies by objecting to the choice of materials, language, and/or participants. However, in so doing, the claim for the utility of flowcharting is being restricted so as not to include some class of situations of which these experiments are instances. The strong form of the flowcharting hypothesis having failed (and the results of Shneiderman et al. have demonstrated that flowcharting is not invariably useful), it is incumbent on those who advocate flowcharts to formulate *and empirically validate* a more limited hypothesis.

The addendum on empirical validation is significant. Even if one's stylistic preference is for debates on programming practice to be carried out on the basis of intuition rather than data, one's intuition in some area is surely somewhat suspect after one's previous claims in that area have failed empirically. Yet, as Shneiderman sadly notes in a retrospective review of this work, "Flowchart critics cheered our results as the justification of their claims, while adherents found fault and pronounced confidence in the utility of flowcharts in their own work" [SHNE80, p. 82]. It is in response to reactions like this that the use of psychology in computer science debates was earlier characterized as "ammunition."

4.2 Indenting (Prettyprinting)

The use of indentation to indicate program structure is another practice for which there is little behavioral evidence. Weissman once again found positive self-evaluations but no performance improvements for either modifying, hand simulating, or answering questions about indented versus unindented versions of programs written in ALGOL W and PL/I [WEIS74]. Shneiderman and McKay (reported in SHNE80) contrasted ability to locate single bugs in indented and unindented versions of two PASCAL programs and found no performance improvement in the indented versions. Love also found no reliable improvement in a reconstruction task for indented and unindented short FORTRAN programs [LOVE77]. This lack of support leads Shneiderman to conclude that, like flowcharting, "the advantage of indentation may not be as great as some believe" [SHNE80, p. 74].

There is, however, a critical difference between flowcharting and indenting concerning the range of phenomena to which it is reasonable to generalize these negative experimental results. "Flowcharting" refers to the use of a supplementary graphical representation of program control flow. As no instance of any such technique has ever been demonstrated to enhance programming performance, it is reasonable to generalize the negative experimental results to the entire class of such techniques. "Indenting," on the other hand, refers to the use of white space formatting to indicate program structure. There is strong reason to believe that such techniques can be effective in some circumstances. One example is the superiority of nested over test-and-jump structures for conditionals discussed earlier. A sweeping conclusion that program layout has no effect would require another explanation for that finding. More intuitively, the most rudimentary use of white space to indicate program structure is the use of a line break between statements in languages such as FORTRAN. Clearly, it would be rash to claim that these studies show that programs in these languages would be as easy to work with if they were laid out without line breaks (or with them placed every *n* characters without reference to the statement structure). The existence of both positive and negative results suggests a search for some set of principles which indicate how and when program formatting techniques will be effective.

Any such principles must be shaped primarily by empirical evaluation rather than by appeals to intuition. However, it seems suggestive that, as opposed to the relative consensus on flowcharting techniques (as reflected, for example, in the existence of standards), program layout schemes are hotly disputed, even within relatively homogeneous programming communities. One can draw one of two conclusions from this lack of agreement. The view of the disputants is that there are decidably good and bad ways of formatting programs. Alternatively, one could conclude that individuals differ as to what attributes they find it useful for the layout to emphasize. This, in turn, could be a function of individual differences, such as personality factors, or, much more likely, simply reflects the formatting scheme that the individual is accustomed to using. Intuitively, the latter seems reasonable because the value of a standardized program layout is in being able to use its (rapidly available) perceptual cues to skip over code that would otherwise have to be at least partially understood as it is scanned. It is unlikely that this would be an effective strategy unless the perceptual operation were highly practiced.

From this point of view indenting is not an inherent attribute of a program. What is indented for one person may seem to be very poorly arranged on the page for another. Any given program formatting scheme might actually degrade the performance of a programmer accustomed to work with programs laid out in another style. At the very least, it seems premature to conclude that indentation is an ineffective technique until it has been demonstrated that neither variation in the formatting style nor practice using some (any) systematic formatting scheme has any effect.

4.3 Variable Naming

There is only scattered research on the oft-claimed benefits of using names with mnemonic properties. Several of Weissman's experiments [WEIS74] addressed this question, but the results are unclear except for the usual improvement of programmer self-evaluation. Shneiderman [SHNE80, pp. 70–72] reports two experiments—a comprehension experiment which showed mnemonic names to be more effective and a debugging experiment that found no such effects. Sheppard et al. [SHEP79] found no evidence that mnemonic names helped professional programmers to memorize ~50 line FORTRAN programs. The overall pattern is unclear. It seems plausible that mnemonic variable names would be less useful in smaller or better understood programs, as they offer less chance for confusion and hence less need for mnemonics. For the same reason one might expect mnemonic names to be more useful for programs with which the programer is less familiar. Such program/programmer variability would account for the inconclusive results, but needs independent experimental confirmation.

4.4 Commenting

The judicious use of comments is almost universally considered to be good practice. Just as for the use of mnemonic variable names, however, empirical evidence for this position is both sparse and equivocal. Weissman found that appropriate comments caused hand simulation to proceed significantly faster, but with significantly more errors [WEIS74]. Shneiderman contrasted students' modification and recall of FORTRAN programs with either high-level (overall program description) or low-level (statement description) comments

[SHNE77a]. The programs with the higher level comments were found to be significantly easier to modify. Sheppard et al. found that neither high- nor low-level comments had any reliable effect on either the accuracy of or the time taken to modify small FORTRAN programs [SHEP79].

Although the evidence for the utility of comments is equivocal, it is unclear what other pattern of results could have been expected. Clearly, at some level comments *have* to be useful. To believe otherwise would be to believe that the comprehensibility of a program is independent of how much information the reader might already have about it. However, it is equally clear that a comment is only useful if it tells the reader something she either does not already know or cannot infer immediately from the code. Exactly which propositions about a program should be included in the commentary is therefore a matter of matching the comments to the needs of the expected readers. This makes widely applicable results as to the desirable amount and type of commenting so highly unlikely that behavioral experimentation is of questionable value.

5. STUDIES OF PROGRAMMING TASKS

Programming "tasks" include activities such as learning a language, debugging, coding, and testing. The work reviewed in this section typically explores a single phase of programming activity in depth, rather than evaluate the effect of some manipulation (such as the effects of a language feature) which might cut across several different phases. Rather than be driven by hypotheses drawn from computing, this work also tends to be more exploratory and descriptive in nature. Perhaps as a result, it is more influenced by psychological theory and is far less normative.

5.1 Learning

Mayer [MAYE75, MAYE81] addressed the question of what students actually learn when learning how to program. He conjectured that students learn the semantics of programming languages by likening their actions to physical or mechanical models, from which they abstract the programming language semantics. Consequently, he reasoned that instruction that provides explicit concrete models (e.g., an erasable chalkboard for memory) for the basic operations of the language would facilitate learning. His 1975 study confirmed this conjecture in the context of teaching BASIC to introductory programming students. More recently [MAYE79], he proposed a detailed decomposition of the knowledge required for BASIC programming into a number of different levels, such as statement, prestatement, or transaction. He further claims that translation between these levels is a critical component of the programmer's skill. However, while this is certainly *one* way to categorize programming knowledge, Mayer has yet to provide any evidence that it is either the only way or the way that programmers actually use.

5.2 Coding

Brooks [BROO77] noted the enormous variation in performance between different programmers of comparable experience for the same tasks (estimates vary between factors of 5 and 100). Such large individual differences, he pointed out, suggest that different programmers might be using quite different strategies, in which case the detailed study of individuals would be more revealing than the aggregated behavior of several individuals as observed in classical experimental studies. He therefore gathered detailed protocols of one experienced programmer working on 23 different programming problems. Using these protocols, he developed a very detailed model of the coding process (the translation from an abstract program to the target computer language) expressed as a production system.

The primary value of such detailed models is that they force the researcher to give a complete account for at least one specific set of actual behavior in all its complexity. For reasonably complex skill, providing any sufficient model at all is a considerable challenge. The amount of information (Brooks estimates $\sim 10^4$ rules) required to account for the detailed behavior in these coding protocols is very compelling. Brooks makes a strong case not only for the necessity of this level of complexity in any descriptively adequate model but also for the necessity of some kind of structure to organize this knowledge.

5.3 Debugging

Debugging is generally considered to be the most expensive phase of software production and has been explored empirically in several studies. Some of the studies discussed earlier that used error detection measures could also be considered studies of debugging.

The earliest systematic behavioral exploration of debugging [YOUN74] contrasted the debugging behavior of novice and expert programmers in several different languages. In addition to a detailed breakdown of the observed errors by the type of the statement in which they appeared, Youngs presents data which indicate that, although both groups made about the same number of errors in their original programs, the experts removed their errors much more quickly. Apparently this is due not so much to the experts' generally superior diagnostic skills, but to their ability to correct the more superficial semantic inconsistencies quickly, so they could concentrate on the more subtle logical ones.

Two studies of expert programmers detecting artificially placed bugs in FORTRAN programs from the IBM Scientific Subroutine Package [GOUL74, GOUL75] classified the bugs by the type of statement in which the bug was placed. Bugs placed in assignment statements were considerably harder to find than those in iterations or array subscripts. This is, however, a somewhat unsatisfying result in that one's intuition is that the "same" bug can often manifest itself in statements of quite different syntactic type, and vice versa. Atwood and Ramsey [ATWO78] further suggest that a variety of quite shallow techniques could have been used to detect many of the array and iteration bugs (e.g., scanning for incommensurate bounds of arrays and indices) but that some of the assignment bugs required a much more complete analysis of the program (including, in some cases, knowledge of its subject matter). They propose instead a representation scheme for the knowledge required to understand such programs and predict that difficulty of debugging will reflect primarily depth of embedding of the buggy component within this representation. They partially confirmed this analysis in a replication of GOUL74, although the interpretation of their findings is clouded by strong differences in the results obtained for the two different programs they used.

6. CRITIQUE AND EVALUATION

This section takes up the question of how compelling a body of research these behavioral studies of programming are, considered both as a guide to computing practice and as a psychological investigation of an interesting complex skill. Unfortunately, as a group they are unsatisfactory in that they are methodologically weak, the effects they report are small, and yet they are presented as if they establish claims that go far beyond their data. These failings can, in turn, be traced to an underlying naive view of programming skill which has been shaped more by the fashions of contemporary computing practice than by any reasonable appreciation of the complexity of the behavior.

6.1 General Methodology

As a body of psychological studies, the research on programming is very weak methodologically. One of the reasons for this is that, as discussed earlier, programming is a very difficult topic on which to do experimental work. Brooks [BROO80] provides a thoughtful review of many of the technical problems associated with carrying out effective research on programming. In particular, he points out the problems associated with selecting participants, test materials, and performance measures, and how one's choices of these can affect one's findings. Brooks also alludes to the established methodological practices of other behavioral sciences and warns that

> Researchers in these other behavioral disciplines have come to expect the use of such tools as an integral part of behavioral research and to judge research, in part, by the skill with which these tools are used. Behavioral research, even if done in a computer science context, will inevitably be judged

> by these same standards of methodological rigor. In order to maintain creditability with behavioral researchers in other areas, behavioral researchers in computer science must pay close attention to methodological issues [BROO80, p. 208].

The real issue, however, is not one of our credibility to social scientists, but the fact that methodological sloppiness introduces artifact and error into both the experimental results and the conclusions that we draw from them.

Apart from the issues raised by Brooks, the empirical research on programming is plagued by a variety of technical flaws of experimental design and analysis. Although some of these have already been discussed in the context of specific studies and further study by study treatment would be tedious, some of them are significant enough to warrant separate discussion.

6.2 Experimental Treatments

The point of an experiment is to generalize to some other (real-world) situation. Consequently, one's experimental treatments should, in some sense, be representative of that other situation both in type and in strength. Excessively forceful or weak experimental treatments limit this generalizability, as they are not representative of conditions to which programmers are subject in the real world. Overly forceful treatments bias results not simply by distorting their magnitude, but by being so salient that participants modify their behavior in response to what they believe the treatment tells them about what the experimenter wants (from which these effects have come to be known as "demand characteristics" [ORNE69]). One commonly used experimental situation which lends itself to this artifact is a classroom experiment in which students are given tests or problem sets on topics which are (or are perceived to be) part of the subject matter of the class. Students in such circumstances are, understandably, very sensitive to any indications as to the "correct" response, and their behavior can easily be determined by this perception rather than by any characteristic of the task. For example, Basili and Reiter [BASI79] contrasted the amount of computer resources used to construct one moderate-size program by structured and unstructured groups in two programming courses. Students in the "structured" condition, however, were being taught techniques whose mastery was *supposed* to result in early error detection and thus reduced computer usage. Therefore, especially since all the students knew that their computer usage was being monitored, it is not at all surprising that those in the "structured" condition held their computer usage to a minimum. The situation demanded that they do so. Needless to say, it does not at all follow either that the structured methodology reduced the amount of "effort" or even that the amount of computer usage would be similarly reduced were these techniques applied in a more neutral setting.

Excessively weak manipulations simply result in no effects being observed. How weak a treatment has to be before it is unrealistic to expect it to have any effect is a matter of judgment. On the other hand, Sheppard et al., noting that less than 10 percent of their participants had had any exposure to FORTRAN77, comment that "... a lack of familiarity with the new constructs may have slowed their debugging. However, we conducted a short training session on these constructs shortly before the experiment, so lack of familiarity was probably not a significant factor" [SHEP79, p. 46]. But what kind of model of programming competence makes it reasonable to expect that such a "short training session" would have any effect on performance?

6.3 Practice Effects

Many of the studies on programming have found strong, persistent practice effects. Some amount of improvement over a series of similar tasks is inevitable as the participants adapt to the experimental environment and task. For that reason it is standard practice to provide a "warm-up" period to allow this transient to dissipate. If the improvement persists strongly beyond such a warm-up period, however, it is clear that the participants are *learning* during the experiment. Generalizing to stable performance from results obtained in a learning phase is dubious, as there is usually no reason to believe either that the different behaviors being contrasted are at similar points on the learning curve or that their relative performance will remain unchanged with further learning. On the other hand, one thing that the existence of strong practice effects *does* establish is that the task is *not* one in which the participants are expert. If it were, they would be well practiced before the experiment began. Specifically, for expert programmer participants, strong practice effects indicate that the experimental task is *not* a substantial component of real-world programming. This is a very serious problem for studies like GREE77 whose main claim for generalizability to real-world programming is that the participants were expert programmers and that the experimental task is "evidently, a substantial component of 'real' programming" [GREE77, p. 108].

6.4 Individual Variability

The high degree of variability among programmers of similar background makes simple experimental designs (in which different participants are used for each condition) prone to negative conclusions, as slight systematic differences between conditions tend to be washed out by large within-condition variation. One of the standard techniques for controlling this is the use of "repeated-measures" designs, in which each participant is observed in more than one condition. However, because of the need to correct for the systematic variance between conditions that share participants, these designs require special analysis techniques. Failure to use these can both create spurious "effects" and mask real ones. Unfortunately, their interpretation is clouded in the presence of strong practice effects, such as those found in some studies of programming (see, e.g., WINE71, pp. 517 ff). Intuitively, this is because in a repeated-measures design each participant is observed in several conditions so that the idiosyncracies of individual participants (such as prior experience) are shared by conditions across which comparisons are made. But if these idiosyncracies are changing during the experiment, this "control" becomes a source of confounds. Counterbalancing techniques may be an effective remedy, depending on the nature of the practice effects, but less is usually known about these than about any other aspect of the behavior.

One of the more striking symptoms of high individual variability is that one occasionally finds a small number of participants whose scores on some measure are far outside the range for the group to which they belong. Such individuals are called "outliers." Their presence not only invalidates the common statistical techniques, but it can both mask real differences (by increasing the variance) and create illusory ones (if they all happen to fall in the same group). Although nonparametric statistics and/or data recoding can be used to avoid the technical problems, it is far preferable simply to discard outliers before the analysis. The reason for this is that very extreme observations strongly suggest that the individual is not typical of those to whom the results are to be generalized. For example, such an individual might be doing something quite different from the other participants, possibly as a result of having misunderstood the instructions. (The classic example is the participant who falls asleep during a reaction time experiment.) Therefore, although examining the outliers separately can often be very illuminating, their data should not be included in standard group analyses. Without the raw data it is impossible to determine to what extent outlier effects might have contaminated the published behavioral studies of programming. However, very few studies make any reference to outliers having been discarded.

6.5 Significance Tests

Another technical issue in experimental analysis concerns the use of statistical tests of significance and the criteria by which one decides whether an observed difference is reliable. Psychology has adopted the standard that the probability of the null hypothesis should be below .05 in order for a researcher to conclude that an observed effect is not simply the result of chance variation. Although one can argue with the choice of this particular value, studies of programming have reported "effects" which have been demonstrated at no better than the $p < .20$ level. The problem with this practice is that significance measures are not estimates of the *size* of an effect, but estimates as to *whether one occurred* at all. Thus raising the p level at which effects are judged to be worthy of explanation has the effect of cluttering the literature with spurious results. The problem is particularly acute in studies which use large numbers of variables. GANN75, for example, reports over 45 tests of significance. BASI79 reports 346! One would expect 69 of these to be significant at the $p < .20$ level, even if none of their manipulations had any effect at all! Nor can one take comfort in the fact that more than 69 such effects were found. It simply raises the question of how many of them were spurious.

6.6 Effect Sizes

By far the most critical problem with the research on programming, related to the misuse of statistical tests of significance, is the weakness of its findings in terms of their size, reliability, and generalizability beyond the experimental contexts in which they were gathered. We have already pointed out that the probability level obtained from a test of significance is not a measure of effect size. Sophisticated data analysis techniques can detect, with any desired level of confidence, minute systematic differences between sets of observations. For this reason methodologists have long urged *all* behavioral researchers to use (or at least report) measures of effect size. They are, however, especially important for applied research, such as that on programming, because the value of some practice is not determined by whether its use is *detectable* but by *how much difference* it makes.

Various different techniques are available to measure the size of an effect [FLEI69]. The most straightforward is the use of ratios or differences between the mean or median values of the different conditions. Although immediate, these measures are weak because they are not invariant under linear transformations of the raw data. Measures of performance are often subject to linear transformation simply by the choice of measure or data collection method. Task completion times, for example, vary substantially with changes in the required error rate, and vice versa. Therefore, unless the data measure is directly interpretable (dollars, for example), it is preferable to measure effect size by the *proportion of the observed variability accounted for* (PVA) by the effect. The merit of this measure is that it indicates to what extent the observed behavior was determined by the experimental manipulation and how much it varied freely. On the other hand, PVA is a very hard standard of evaluation. Human behavior is sufficiently complex that even the most reliable psychological phenomena rarely account for very large proportions of the variance.

The weakness of PVA as a measure of effect size is not a question of the details of the experiment itself but of what class of situations the experiment is considered to represent. One can vary the PVA in an experiment simply by varying the amount of extraneous variance. A tightly controlled experiment in which very few factors are allowed to vary will produce much higher PVAs than one in which extraneous factors vary freely. The evaluation of an experimental PVA thus involves determining (or judging) the relative importance of the factors that were controlled and those that were left free to vary. Then there is the separate question of how significant (in terms of practical value) a given real-world PVA is considered to be.

Virtually none of the studies on programming report either PVA estimates of effect size (SHEP79 is an exception) or the information which would allow them to be computed. Even when expressed as mean differences or ratios, the effects reported are generally weak. When one further considers how well they would survive the introduction of other sources of variation, the outlook is bleak indeed. Many of them did not! Of those that did, we have already commented on the size of some of them relative to "incidental" effects such as practice. The apogee of control over generalizability is reached in the Sime et al. studies on conditionals [SIME73, SIME77]. These were carried out using experimental tasks that involved "programming" a mechanical rabbit because their nonprogrammer participants were so intimidated by the idea of interacting with a computer that their nervousness dominated any effects of notation. The conclusion seems inescapable that the notation for expressing conditionals can hardly be a leading term in novice programming performance.

6.7 Theoretical Orientation

One of the most salient characteristics of the psychological research on programming is its preoccupation with the issues of contemporary computing practice. While the practical concern is understandable, many of the studies are so narrowly focused in an attempt to settle some debate among computer scientists that they are of dubious scientific value. GANN75 is the most extreme example. Varying nine independent factors simultaneously is an absurd way to do empirical research and it is inconceivable that such a design could have come from any motivation other than to make a point to the programming community. Unfortunately, the demonstration failed. As a scientific experiment it could never have succeeded.

Another consequence of this dependency on computing is that behavioral researchers tend, possibly in an attempt to make their work appeal to computer scientists, to generalize far beyond their data. We have already had cause to remark on several instances of this. The most damaging, because it will undoubtedly influence many computer scientists' ideas about behavioral research, is Shneiderman's review [SHNE80]. Detailed discussions of experimental results are interleaved with totally (empirically) unsupported opinions on programming style (e.g., sections 4.3.2 and 4.3.3 and the summaries on pages 77 and 81). Much of this material would be quite legitimate, intuitively based argument in a computer science debate. However, its presentation as part of a discussion of empirical research completely blurs the distinction between data and intuition, inviting readers to reject data that do not support their preconceptions. This makes the entire empirical enterprise moot.

6.8 Summary

Methodological critiques are of limited appeal. The reason for this one is that most of this work has never been critically reviewed *as behavioral research.* Considered as such, it has serious flaws. Unfortunately, much of it has been taken at face value and is cited, in one-sentence summary form, as having established positions for which the studies provide only slight evidence when examined closely. The absence of a critical review process, coupled with the very considerable difficulty of research in this area and the constant tendency to drift into intuitively based argument and generalize far beyond what has been established, has created a pseudopsychology of programming. At one time that might have been healthy—after all, the bulk of Weinberg's book was simply a plea for computer scientists to *consider* behavioral issues. Now, however, most computer scientists are quite sophisticated armchair psychologists. It is therefore appropriate that these psychological discussions now be established on a rather more solid footing.

7. A CHARACTERIZATION OF PROGRAMMING SKILL

The unimpressive results of behavioral research on programming could simply be the results of sloppy methodology, of a poor choice of hypotheses from computer science, and of the considerable practical difficulty of investigating complex behavior. While all of these have had their impact, the basic problem is a fundamental misunderstanding of the nature of programming skill. Specifically, most psychological research on programming assumes (usually implicitly) that different programming tasks (produced, for example, by differences in notation or practice) vary in difficulty and that the level of difficulty is an attribute of the task. Further, the motivation for (and generalizations made from) much of this work is the belief that the difficulty of large tasks is an aggregation of the difficulties of many component tasks, such as inverting a logical predicate when reasoning backward through an ELSE clause. These assumptions have formed the basis for many successful human factors investigations for a variety of different skills and are thus certainly a reasonable starting point in the absence of more specific knowledge. The role of the psychologist, in this view of the world, is to evaluate the difficulties of different aspects of the programming task so that the more difficult can be eliminated from programming systems.

The problem is that these assumptions are simply false for programming. Most notably, they give no account of the most salient single fact about programming, which is that the difficulty of programming is a very nonlinear function of the size of the problem. The primary requirement of any psychological account of programming is that it give an account of this nonlinearity. However, the simple aggregation of difficulty model provides no mechanism by which such a nonlinearity could be generated. Further, no program of research predicated on a simple composition of component task difficulties is likely to even consider this question.

More fundamentally, programming is clearly a learned skill, and, therefore, what is easy or difficult is much more a function of what skills an individual has learned than of any inherent quality of the task. While some of the lower level component behaviors may vary reliably in difficulty across individuals, most programming behavior is dominated by higher level skills and knowledge whose plasticity simply does not allow such reliable differences. This simple observation casts the existing psychological research on programming in a completely different light. High individual variances, strong practice effects, and (consequently) weak findings are exactly what one would expect from studying the average performance of highly learned skills across diverse collections of individuals. Studying naive or semitrained programmers in this way does not avoid this problem but merely measures the common tendencies that result from the use of shared cultural skills. In the course of learning to program, however, these are rapidly displaced by the special-purpose skills that constitute expertise in a particular environment. To characterize the (slight) differences found between groups of novices as deep principles of programming skill (or major sources of variance in expert performance) is simply foolish.

The evidence for a characterization of programming as a "learned skill" is mainly negative, as such a characterization says more about what a skill is not than about what it is. Experimental demonstrations of the plasticity of programming performance with training would provide direct support. However, the most compelling subjective demonstration for the experienced programmer is to introspect (an old psychological technique now fallen into disfavor) while formulating a procedure which, given a set of N positive numbers, finds the largest. Nearly all experienced programmers will immediately produce the following procedure, notational variants aside.

```
$m \leftarrow 0$
**for** $i$ **from** 1 **to** $N$ **do** (**if** $a_i > m$ **then** $m \leftarrow a_i$)
```

Consider how that procedure could have been generated or understood. One way, the way that you might believe it was done if you read the literature on programming that is written by computer scientists, is by formulating a loop invariant. For this loop the appropriate invariant is that, at the end of the kth pass through the loop, $m \geq a_j$ for $j \in [1, k]$. Once formulated, that invariant can be proved by induction. Having been proved by induction, it can be instantiated for $k = N$ and now constitutes a proof that m is a maximum of the set, which is the desired result.

The problem is that nobody does it that way. And the reason nobody does it that way, except in introductory programming courses, is that it takes too long and is far too complicated. If you know how to program, you would neither generate this program nor synthesize an understanding of it. You would *know* the answer. You would *recognize* the problem, key directly into that knowledge, and pull out a working procedure. The compelling subjective evidence for this, alluded to above, is the complete absence of any introspective trace whatsoever!

The immediacy with which the expert programmer "solves" problems of this sort indicates that the programmer's expertise is made up of an enormous number of interrelated pieces of knowledge. The primary piece of direct behavioral evidence for this position is Shneiderman's replication [SHNE76] for programming of Chase and Simon's classic study on memory for chess positions [CHAS73]. In both these studies it was found that experts in a particular domain could memorize information from that domain (i.e., a program or a chess position) far better than novices, provided that the information was appropriately structured. If the structure were made random (by shuffling the statements of the program or rearranging the chess pieces), the advantage of the expert would be greatly reduced. The standard interpretation of this result is that the expert has no better memory than the novice, but rather an elaborate knowledge structure in terms of which correspondingly structured items can be very efficiently encoded. Further support for the notion of a large knowledge base comes from Brooks' study of coding discussed earlier [BROO77], in which he estimated that ~10^4 elementary rules would be required to capture the knowledge used in his protocols.

While there is currently little evidence as to how these knowledge structures are organized, some recent work in automatic programming based on "natural deduction" techniques (e.g., BARS79, RICH78) and programming methodology (e.g., FLOY79) is suggestive. This work shares the notion that programming knowledge can be thought of as a collection of units ("frames," "paradigms," "schemata"), each of which is organized as a program fragment, abstracted to some degree, together with a set of propositions about its behavior and rules for combining it with others, and indexed in terms of the problem classes for which it is appropriate. The *maximum* procedure discussed above, if not stored directly, can be derived by (or described as) a single transformation of a *reduction* schema (so called after the APL operator) or a two-step transformation of various *bounded iteration* schemata. The descriptive economy and structuring power of this basic idea make it attractive to programming theorists. There is currently no direct empirical evidence for it as a behavioral theory of programming. On the other hand, it does provide a theory which is plausible and sufficient (in that the automatic programming systems which represent their knowledge of programming this way demonstrate that this representation can be used to write programs) and can account for the nonlinear growth of difficulty with size in terms of the behavior of the appropriate deduction algorithms.

However programmers' knowledge bases are actually organized, their existence and size seem clear. Hypotheses which posit differences in either individual aptitude or task difficulty are therefore, at best, extremely difficult to investigate, as the enormous size of the knowledge bases being drawn on imply that different individuals approach the "same" task with vastly different resources. Comparing their behavior is like contrasting the work of two tile layers, each of whom has covered an equivalent wall, working with radically different sizes and colors of tiles. Similarity of pattern is unlikely.

8. HOW TO PROCEED

The diversity of goals which motivate the behavioral exploration of "programming" suggests that programming may be an equally diverse collection of skills. Most studies of programming difficulty are prefaced, by way of justification, with allusions to the "software crisis" or the need for widerspread procedural literacy. These, however, are quite different concerns, much better addressed by targeted research than by a vague assault on the "difficulty of programming."

If one's concern is the software crisis and the rapidly increasing cost of program development, the appropriate strategy is to determine whence that cost is coming, rather than assume that it is the performance of the individual programmer. Determining the source of the cost requires detailed studies of large programming projects—a job for a "cost anthropologist," not a psychologist. Such studies are expensive, difficult, and subject to the caveats mentioned earlier that apply to all *in situ* studies, which is why so little work of this type has been done.

On the other hand, if one's concern is procedural literacy, the key question is the determinants of the learning behavior of novices. There is no reason to believe that these have anything to do with the determinants of expert performance. As novices do not have the specialized knowledge and skills of the expert, one might expect their performance to be largely a function of how well they can bring their skills from other areas to bear [SHEI80]. MAYE75 can be interpreted as providing some support for this point of view.

Finally, given that the performance of the expert programmer is the topic of interest, the appropriate approach at this stage of our knowledge is the detailed study of individual expert performance (in the style of BROO77) in an attempt to identify the components of expert skill. Our primary need at the moment is for a theory of programming skill that can provide both general guidance for system designers and specific guidance to psychologists selecting topics for detailed study. The experimental investigation of such factors as the style of conditional notation is premature without some theory which gives some account of why they might be significant factors in programmer behavior. The existing literature of attempts to provide complete performance models of complex skills (e.g., CARD80, BROW78) suggests that even broad theories will have to be very much more complex than the simplistic hypotheses that have guided the work on programming so far.

ACKNOWLEDGMENTS

Many people have helped shape this paper. In particular, I would like to express my gratitude to Ron Kaplan, Joanne Martin, and the referees for their comments; to Tom Moran, the editor of this special issue, for both detailed commentary and extraordinary editorial patience; and to my colleagues at Cognitive and Instructional Sciences at Xerox PARC for the supportive, yet critical, environment without which topics of this complexity would seem more intimidating than exciting.

Table 1 and Figure 1 are reprinted with permission from M. E. Sime, T. R. G. Green, and D. J. Guest, "Scope marking in computer conditionals—A psychological evaluation," *Int. J. Man-Mach. Stud.* **5**, (1973). Copyright by Academic Press, Inc. (London) Ltd. Table 2 is adapted from T. R. G. Green, "Conditional program statements and their comprehensibility to professional programmers," *J. Occup. Psychol.* **50** (1977). Table 3 is reprinted from H. C. Lucas and R. B. Kaplan, "A structured programming experiment," *Comput. J.* **19** (1976).

REFERENCES

ATWO78 ATWOOD, M. E., AND RAMSEY, H. R. "Cognitive structures in the comprehension and memory of computer programs: An investigation of computer debugging," Tech. Rep. TR-78-A21, U.S. Army Research Institute for the Behavioral and Social Sciences, Alexandria, Va., 1978.

BARS79 BARSTOW, D. *Knowledge based program construction,* North-Holland, Amsterdam, 1979.

BASI79 BASILI, V. R., AND REITER, R. W. "An investigation of human factors in software development," *Computer* **12** (1979), 21-40.

BROO75 BROOKS, F. P. *The mythical man-month,* Addison-Wesley, Reading, Mass., 1975.

BROO77 BROOKS, R. E. "Towards a theory of the cognitive processes in computer programming," *Int. J. Man-Mach. Stud.* **9** (1977), 737-751.

BROO80 BROOKS, R. E. "Studying programmer behavior experimentally: The problems of proper methodology," *Commun. ACM* **23**, 4 (April 1980), 207-213.

BROW78 BROWN, J. S., AND BURTON, R. R. "Diagnostic models for procedural bugs in basic mathematical skills," *Cognitive Sci.* **2** (1978), 155-192.

CAMP79 CAMPBELL, D. T., AND COOK, T. D. *Quasi-experimentation: Design and analysis for field settings,* Rand McNally, Chicago, Ill., 1979.

CARD80 CARD, S. K., MORAN, T. P., AND NEWELL, A. "Computer text editing: An information processing analysis of a routine cognitive skill," *Cognitive Psychol.* **12** (1980), 32-74.

CHAS73 CHASE, W. G., AND SIMON, H. A. "Perception in chess," *Cognitive Psychol.* **4** (1973), 55-81.

DIJK68 DIJKSTRA, E. W. "GOTO statement considered harmful," *Commun. ACM* **11**, 3 (March 1968), 147-148.

DIJK76 DIJKSTRA, E. W. *A discipline of programming,* Prentice-Hall, Englewood Cliffs, N.J., 1976.

FLEI69 FLEISS, J. L. "Estimating the magnitude of experimental effects." *Psychol. Bull.* **72** (1969), 273-276.

FLOY79 FLOYD, R. W. "The paradigms of programming," *Commun. ACM* **22**, 8 (Aug. 1979), 455–460.

GANN75 GANNON, J. D., AND HORNING, J. J. "The impact of language design on the production of reliable software," *IEEE Trans. Softw. Eng.* **SE-1** (1975), 179–191.

GANN76 GANNON, J. D. "An experiment for the evaluation of language features," *Int. J. Man-Mach. Stud.* **8** (1976), 61–73.

GANN77 GANNON, J. D "An experimental evaluation of data type conventions," *Commun. ACM* **20**, 8 (Aug. 1977), 584–595.

GOUL74 GOULD, J. D., AND DRONGOWSKI, P. "An exploratory investigation of computer program debugging," *Hum. Factors* **16** (1974), 258–277.

GOUL75 GOULD, J. D. "Some psychological evidence on how people debug computer programs," *Int. J. Man-Mach. Stud.* **7** (1975), 151–182.

GREE77 GREEN, T. R. G. "Conditional program statements and their comprehensibility to professional programmers," *J. Occup. Psychol.* **50** (1977), 93–109.

KERN74 KERNIGHAN, B. W., AND PLAUGER, P. J. *The elements of programming style*, McGraw-Hill, New York, 1974.

LOVE77 LOVE, T. "Relating individual differences in computer programming performance to human information processing abilities," Ph.D. dissertation, Univ. Washington, 1977.

LUCA76 LUCAS, H. C., AND KAPLAN, R. B. "A structured programming experiment," *Comput. J.* **19** (1976), 136–138.

MAYE75 MAYER, R. E. "Different problem solving competencies established in learning computer programming with and without meaningful models," *J. Educ. Psychol.* **67** (1975), 725–734.

MAYE79 MAYER, R. E. "A psychology of learning BASIC," *Commun. ACM* **22**, (Nov. 1979), 589–593.

MAYE81 MAYER, R. E. "The psychology of learning computer programming by novices," *Comput. Surv.* **13** (March 1981), 121–141.

ORNE69 ORNE, M. T. "Demand characteristics and the concept of quasi-controls," in *Artifact in behavioral research*, R. Rosenthal and R. L. Rosnow, Eds., Academic Press, New York, 1969.

RICH78 RICH, C., AND SCHROBE, H. "Initial report on a Lisp programmer's apprentice," *IEEE Trans. Softw. Eng.* **SE-4** (1978), 456–467.

SACK70 SACKMAN, H. *Man-computer problem solving*, Auerbach, Princeton, N.J., 1970.

SHEI80 SHEIL, B. "Teaching procedural literacy," in *Proc. ACM Annual Conf.*, 1980, pp. 125–126.

SHEP79 SHEPPARD, S., CURTIS, B., MILLIMAN, P., AND LOVE, T. "Modern coding practices and programmer performance," *Computer* **12** (1979), 41–49.

SHNE76 SHNEIDERMAN, B. "Exploratory experiments in programmer behavior," *Int. J. Comput. Inf. Sci.* **5** (1976), 123–143.

SHNE77a SHNEIDERMAN, B. "Measuring computer program quality and comprehension," *Int. J. Man-Mach. Stud.* **9** (1977), 465–478.

SHNE77b SHNEIDERMAN, B., MAYER, R., MCKAY, D., AND HELLER, P. "Experimental investigations of the utility of detailed flowcharts in programming," *Commun. ACM* **20**, 6 (June 1977), 373–381.

SHNE79 SHNEIDERMAN, B., AND MAYER, R. "Syntactic/semantic interactions in programmer behavior: A model and experimental results," *Int. J. Comput. Inf. Sci.* **7** (1979), 219–239.

SHNE80 SHNEIDERMAN, B. *Software psychology*, Winthrop, Cambridge, Mass., 1980.

SIME73 SIME, M. E., GREEN, T. R. G., AND GUEST, D. J. "Psychological evaluation of two conditional constructions used in computer languages," *Int. J. Man-Mach. Stud.* **5** (1973), 123–143.

SIME77 SIME, M. E., GREEN, T. R. G., AND GUEST, D. J. "Scope marking in computer conditionals—A psychological evaluation," *Int. J. Man-Mach. Stud.* **9** (1977), 107–118.

WEIN71 WEINBERG, G. M. *The psychology of computer programming*, Van Nostrand Reinhold, New York, 1971.

WEIS74 WEISSMAN, L. "A methodology for studying the psychological complexity of computer programs," Ph.D. dissertation, Univ. Toronto, Canada, 1974.

WINE71 WINER, B. J. *Statistical principles in experimental design*, McGraw-Hill, New York, 1971.

YOUN74 YOUNGS, E. A. "Human errors in programming," *Int. J. Man-Mach. Stud.* **6** (1974), 361–376.

RECEIVED JANUARY 1981; FINAL REVISION ACCEPTED JANUARY 1981