Int. J. Man–Machine Studies (1993) **38**, 985–997

Why can't smart students solve simple programming problems?

RUSSELL L. SHACKELFORD AND ALBERT N. BADRE

College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA

(Received 14 May 1991 and accepted in revised form 28 June 1992)

Computer programming education is evolving on several fronts. This study investigated aspects of the question: how adequate are our conceptions of how to best teach the subject matter? Previous research suggested that the teaching of programming should be focused on "problem solving strategies" ("Model A") rather than on the syntactic/semantic aspects of writing programs ("Model B"). This study was designed to test for student programming performance differences based upon feedback content obtained from each of these two models. The results indicate that certain Pascal loop construct definitions are too general with respect to loop construction. A stricter loop schema resulted in superior performance. The findings also argue that the WHILE construct should receive stricter instructional treatment. In addition, a "constructive" approach to "Model A" feedback (focusing on programmer processes) correlated with subsequent performance superior to that of the control group, whereas neither the conventional approach to "Model A" feedback nor "Model B" feedback (focusing on programmer errors) did so. This finding argues that an approach which relies on narrative treatments and/or error messages is ineffective and that constructive decision rules should serve as a basis for feedback generation and perhaps other aspects of teaching.

1. Introduction

A cursory look at research findings shows that students in introductory programming courses produce correct Pascal programs to solve simple problems *less than half of the time* (Soloway *et al.* 1982; Soloway, Bonar & Ehrlich, 1983; Soloway & Ehrlich, 1984; Shackelford, 1989). Why is this so?

Problem difficulty is not a factor. In studies at Yale (Soloway *et al.*, 1982) and at Georgia Tech (Shackelford, 1989), subjects in introductory programming courses were given simple averaging tasks; fewer than half provided correct Pascal solutions. Averaging is generally taught in the 5th grade. It is not reasonable to assume that undergraduates at Yale and Georgia Tech were taxed by a kind of problem routinely taught to 11 year old children.

We can also dismiss the difficulty of algorithmic thinking as a factor. Creating algorithms to average a series of numbers is analogous to creating other kinds of procedural directions. Creating procedural directions or "recipes of action" is evident in numerous everyday activities, such as providing directions for driving to one's home or telling someone how to prepare chocolate chip cookies. Such tasks are routinely performed by teenagers across wide levels of intelligence and education. It is not reasonable to assume that subjects from Yale and Georgia Tech

0020-7373/93/060985 + 13$08.00/0

were taxed by the procedural complexity of adding a series of integers, then computing their average.

Finally, we can dismiss the inherent difficulty of the programming language in which the solutions were to be expressed. While subjects made various syntactic errors, there was no widespread pattern of them. In fact, like Spohrer and Soloway (1986), we found nothing to indicate that subjects misunderstand the language elements or constructs. To the contrary, the most complex constructs called for by the problems, the Pascal looping constructs, were almost always implemented successfully with respect to both syntax and semantics.

To what, then, can we attribute the consistently poor performance of such bright students on such simple problems? Our research suggests that it is the consistently incomplete and, in certain respects, *inaccurate* instructional treatment of language construct *usage* that bears much of the responsibility for the widespread pattern of failure. In particular, we believe that we have identified specific and potentially very serious problems in the conventional instructional treatment of the Pascal iteration constructs. We have also identified solutions to these problems.

2. Background: the "looping strategy" issue

Research by Soloway *et al.* (1982) defined the concept of "looping strategy" and investigated certain aspects of its role in novice programming performance. "Looping strategy" refers to the ordering of "Get value" and "Process value" steps within the loop body. There are two varieties: the "Get-Process" strategy and the "Process-Get" strategy. Example A, below, exemplifies a "Get-Process" strategy by virtue of the READ statement preceding the processing step within the loop body. Example B exemplifies the "Process-Get" strategy, as it features the in-loop READ step following the in-loop processing step.

Example A:

```
Repeat
  Read(Data);
  Sum:=Sum+Data
until (Sum>100);
```

Example B:

```
Read(Data);
While (data<>9999)
begin
  Sum:=Sum+Data;
  Read(Data)
end; (while)
```

Soloway *et al.* (1982) presented subjects with the three simple averaging problems in Figure 1. They created each problem explicitly to invite one of the three Pascal looping constructs: Problem 1 is the kind of problem for which a FOR loop is useful, Problem 2 presents a situation for which the REPEAT loop is optimal, and Problem 3 is best solved by means of a WHILE loop.

Problem 1.
Write a program which reads 10 integers and then prints out the average. Remember, the average of a series of numbers is the sum of those numbers divided by how many numbers there are in the series.

Problem 2.
Write a program which repeatedly reads in integers until their sum is greater than 100. After reading 100, the program should print out the average of the integers entered.

Problem 3.
Write a program which repeatedly reads in integers until it reads the integer 99999. After seeing 99999, it should print out the correct average. That is, it should not count the final 99999.

FIGURE 1. Three simple looping problems.

For each of these three problems, when a solution is programmed using the preferred loop construct, a given strategy follows. Thus, a FOR loop solution to Problem 1 will naturally present a "Get-Process" strategy, as will a REPEAT loop solution to Problem 2. A WHILE loop solution to Problem 3 will naturally evidence a "Process-Get" strategy.

For each of the three problems, a correct solution can be constructed which does not use either the preferred construct or the preferred strategy as given above. Thus, correctness is not dependent upon the preferred choices of either construct or strategy. Soloway *et al.* (1982) investigated whether construct or strategy choices would prove to be predictors of success. They found that neither was a success predictor in general, but that appropriate strategy choice was a success predictor for Problem 3, the problem which calls for the WHILE loop with a "Process-Get" strategy. That is to say, the use of a WHILE loop for Problem 3 did not reliably anticipate success but the use of a "Process-Get" strategy did.

Soloway, Bonar & Ehrlich (1983) found that subjects preferred to use a looping construct which allowed the use of the "Get-Process" strategy to one which implied the "Process-Get" strategy. They concluded that the "Get-Process" strategy is natural to people while the "Process-Get" strategy is not.

In summary, Soloway and colleagues (a) defined the issue of in-loop Get/Process ordering, (b) demonstrated that it is sometimes relevant to novice programmer performance (for a specific kind of problem), and (c) demonstrated that it is relevant to programmer preferences with respect to problem solving which involves looping.

3. Purpose and design of the study

Given that the ordering of in-loop "Get" and "Process" steps is in some ways relevant to novice programming and problem solving, we were curious as to whether guidance with respect to Get/Process ordering would have an effect on subsequent novice programmer performance. Experience with subjects in pilot studies led us to pursue means by which we could guide novices to appropriate Get/Process ordering decisions without explicitly educating them about the "looping strategy" concept.

We chose to attempt this by a two-part intervention: (a) instructing subjects to

```
FOR (test)          REPEAT              Get a value
begin                 Get a value       WHILE (test)
  Get a value         Process value     begin
  Process value     UNTIL (test)          Process value
end (for)                                 Get a value
                                        end (while)
```

FIGURE 2. Order-influenced loop schema definitions.

comply with redefined Pascal loop schema that are more strict with respect to the ordering of "Get" and "Process" statements, and (b) providing subjects with decision rules by which they could readily select the appropriate redefined loop schema.

The redefined loop schema are presented in Figure 2. They simply formalize what is found by observation: when a given loop construct is deployed in a solution to a problem for which that construct is preferred, a given ordering of "Get" and "Process" steps follows. Thus, we redefine the FOR and REPEAT constructs to imply a "Get-Process" ordering and the WHILE construct to imply a "Process-Get" ordering.

Two versions of a loop-selection decision rule were developed (see Figure 3). The "descriptive behavior rule" is simply an articulation of the rule that is implied in many introductory programming texts, and is labeled "descriptive behavior" because its last two clauses are based on descriptions of the behavior of REPEAT and WHILE loops. Consideration of this rule led us to question its "followability": it demands that the user make a determination about the behavior of the loop which presumably does not yet exist. Thus, this rule presents the novice with a circular problem: "*first choose a construct so you can write the loop, but first write the loop so you can judge its behavior and then choose a construct*".

In response to this perceived problem, we developed the "constructive use rule" to correct the circularity of the descriptive behavior rule. It is intended to focus attention on the control variable. By "control variable", we mean the variable whose value controls whether the loop continues or terminates. In a FOR loop for

The descriptive behavior rule:

(a) if the number of passes through the loop is known in advance, use a "For" loop;
(b) if it might be that no passes through the loop will occur, use a "While" loop;
(c) if at least one pass through the loop will occur, use a "Repeat" loop.

The constructive use rule:

(a) if the value of the control variable is simply a count of the number of iterations, use a "For" loop;
(b) if the value of the control variable exists apart from the loop (you need only access it), use a "While" loop;
(c) if the value of the control variable exists only after computation within the loop, use a "Repeat" loop.

FIGURE 3. Two decision rules for construct selection.

Problem 1, the control variable will be a counter used to keep track of the number of passes through the loop, e.g., "*n*" in "FOR $n = 1$ to 10 DO". In a REPEAT loop for Problem 2, the control variable will be a variable used to keep track of the sum of the integers that have been read, e.g., "sum" in "UNTIL (sum > =100)". In a WHILE loop for Problem 3, the control variable will be the variable into which the various input integers are read, e.g., "read_val" in "WHILE (read_val < >9999) DO".

The constructive use rule allows the programmer to base his choice of loop construct on *what must be done* to obtain the value of the control variable. If the control variable is nothing more than a counter variable that needs to be checked, then a FOR loop is indicated. If the value of the control variable must be computed within the loop, then a REPEAT loop is indicated. If the value of the control variable exits elsewhere, i.e., if it is not *computed* within the loop but rather is *fetched* by the loop, then a WHILE loop is indicated.

We devised this rule, reasoning that novices would be better able to consider certain properties of the value of the control variable than they would be able to consider the behavior of a loop which does not yet exist. It is labeled "constructive use" due to our perception that it might better allow novices to effectively construct a loop based on *data available to them at the time of loop construction.*

The constructive use rule is adequate for decisions regarding three of four common iteration problems. It does not address the "*n* and a half" loop (Dijkstra, 1973) which involves processing steps both before and after the test. Pascal lacks a construct to directly address such a problem; with Pascal, such loops are best implemented via the WHILE construct and some rather convoluted logic. Such loops are directly addressed by the "test-and-exit in the middle of the loop" construct which is absent from Pascal but provided by Ada (Pyle, 1981).

The three problems from Figure 1 were given to 87 subjects in the sequence 1, 2, 3. We used this sequence to allow comparison with the earlier findings of Soloway *et al.* (1982) who used the same sequence. For each problem, subjects were instructed to develop a Pascal program to solve the problem. Solutions were developed using a text editor; no compilation or runtime facilities were provided. Programs were judged to be correct if they would compile and product the correct answer.

For purposes of our investigation, we define "novices" as those undergraduates who are students in an introductory programming course. Novice subjects were assigned to four experimental groups such that each group was equally weighted with respect to both SAT score and pre-test performance; Problem 1 served as pre-test. Problem 2 served as the experimental intervention task. The four groups were treated differently following Problem 2 performance: Group 1 received the constructive use rule and schema definition instructions; Group 2 received the descriptive behavior rule and schema definition instructions; Group 3 received conventional grader-generated error messages in response to their Problem 2 performance; Group 4 served as control, receiving no feedback or special instructions. Following the differential treatment of the four groups, Problem 3 was assigned to all subjects. It served as the post-test: intergroup performance was evaluated based on Problem 3 performance. Processing of subject responses was performed with the help of the *OPTIMUS Teaching Information System.* χ^2 and Dunn–Bonferroni procedures were used in statistical analysis.

TABLE 1
Group performance by program correctness

	Problem 1 pretest	Problem 2 intervention	Problem 3 post-test
Group 1 ($n = 22$) (Schema + c.u. rule)	7	9	15
Group 2 ($n = 22$) (Scheme + d.b. rule)	7	8	12
Group 3 ($n = 21$) (Error messages)	6	6	6
Group 4 ($n = 22$) (Control group)	7	9	7

4. Results

Performance of the four groups in terms of program correctness is summarized in Table 1. The differential treatment of the groups resulted in significantly different levels of subsequent post-test performance ($\chi^2 = 9{\cdot}35$, $df = 3$, $p < 0{\cdot}05$). This difference was caused by the schema definition instructions resulting in performance superior to and significantly different from both error messages and control treatments ($p < 0{\cdot}05$). Coupled with schema definition instructions, the constructive use rule resulted in performance superior to and statistically distinguishable from the error message and control treatments; the descriptive behavior rule did not.

Performance of the four groups in terms of construct-selection correctness is summarized in Table 2. The constructive use rule was clearly more "followable" than was the descriptive behavior rule: subjects receiving this decision rule chose the preferred construct more often than did those who received the descriptive behavior rule ($p < 0{\cdot}05$). The descriptive behavior rule had no apparent impact on construct selection; construct choice resulting from the descriptive behavior rule was inferior to and not statistically distinguishable from error message and control treatments. It appears that the descriptive behavior rule is simply not followable by novices.

For all trials, our findings parallel those of Soloway *et al.* (1982) on some key points: neither construct nor ordering choice proved to be a predictor of success in general, while ordering choice did prove to be a success predictor for Problem 3. Our findings differ from theirs in that we found construct choice to be a success predictor for Problem 2.

4.1. ANALYSIS OF STUDENT PERFORMANCE

Of 261 responses, only 44 were generated after exposure to the schema definitions. We examined all responses across all trials for conformity with the schema definitions and found that subjects naturally conformed to two of the three order-strict loop schema. All of the 36 FOR loops conformed to the preferred "Get-Process" ordering; all but two of the 58 REPEAT loops (96·6%) did the same.

TABLE 2
Performance by correct selection of construct

	Correct	*n*	Percent
Constructive use rule	18	22	81·8
Descriptive behavior rule	12	22	54·5
No rule	24	43	55·8

Thus, it appears that it simply does not occur to novices to use the "Process-Get" ordering with the FOR and REPEAT loops. It is only in the case of WHILE loops that the schema definitions become an issue: only 33 of the 133 WHILE loops (24·8%) conformed to the preferred "Process-Get" ordering.

Performance with respect to the WHILE construct is particularly striking. Fully 3/4 of WHILE loop implementations featured the "Get-Process" WHILE loop. This particular construct/ordering combination occurred frequently across all three problems and was clearly the favorite choice of subjects, accounting for 44% of *all* legal loop implementations. Its success rate was by far the worst of all construct/ordering combinations. When we separated all trials into two categories, those that used the "Get-Process" WHILE and those that used *any other* construct/ordering combination, we found that the "Get-Process' WHILE's 21·0% success rate compared to a success rate of 57·0% for all other loop attempts. This is a very powerful finding: *overall success rate improves by a factor of nearly 3 if we simply remove all programs with "Get-Process' WHILE implementations,* a difference that is significant at the 0·0001 level.

It is not the WHILE construct *per se* that was troublesome: programs featuring "Process-Get" WHILE loops had a 69·7% success rate, the highest of any construct/ordering combination. It was only the "Get-Process" WHILE that was failure prone. It proved to be the only reliable predictor of failure.

The results clearly indicate that (a) novices perceived the "Get-Process" WHILE loop as the most widely applicable of all construct/ordering combinations, (b) programs which utilized it were prone to failure, and (c) when instructions which prohibited its use were applied, subsequent performance improved significantly.

TABLE 3
Construct/schema success ratios

	FOR		REPEAT		WHILE	
	"G-P"	"P-G"	"G-P"	"P-G"	"G-P"	"P-G"
Problem 1	16/36	0/0	0/4	0/0	3/26	2/3
Problem 2	0/0	0/0	18/28	0/0	13/49	1/2
Problem 3	0/0	0/0	14/24	1/2	5/25	20/28
Total	16/36	0/0	32/56	1/2	21/100	23/33
Overall	44·4%	NA	57·1%	50·0%	21·0%	69·7%

NA, not applicable.

These findings indicate that the conventional order-blind WHILE loop invites novice failure.

4.2. ORDER CLASH: A MAJOR CAUSE OF FAILURE

Regardless of Get/Process ordering, the WHILE construct was the favorite choice of the population. Of all responses, more than half featured WHILE loops (133 of 261). Why is this so? The most obvious answer is that students are taught to prefer the WHILE. In all of the 27 textbooks of programming in Pascal which we examined (Findlay & Watt, 1978; Grogono, 1978; Welsh & Elder, 1979; Holt & Hume, 1980; Conway, Gries & Zimmerman, 1981; Schneider & Bruell, 1981; Cooper & Clancy, 1982; Richards, 1982; Schneider, Weingart & Perlman, 1982; Dale, 1983; Gear, 1983; Graham, 1983; Pollack, 1983; Skvarcius, 1984; Nanney, 1985; Jones 1986; Koffman, 1986; Lamb, 1986; Ledgard, 1986; Leestma & Nyhoff, 1986; Nance, 1986; Peters, 1986; Savitch, 1986; Smith, 1986; Walker, 1986; Wells, 1986; Wetzel & Bulgren, 1986) the WHILE loop is given the most attention and is positioned as the most powerful and generalizable construct. None of the examined texts indicated that the WHILE is *ever* seen as contraindicated. Our findings lead us to conclude that such a positioning of the WHILE is wrong. However, given that it is the accepted norm, it is perfectly natural that students would select the WHILE in the absence of compelling reasons to do otherwise. They are, in effect, taught to "trust" the WHILE across the entire spectrum of possibilities and our subjects seem to have learned this lesson well.

Given that students are effectively taught to rely on the WHILE, why the propensity for implementing it with a "Get-Process" ordering? In our study, they chose to do so by a margin of 3 to 1. There is an obvious explanation: the "Get-Process" ordering is most natural to people as they consider iterative behavior. When a task requires repeatedly getting something then processing it, it is logically necessary to get an item before acting upon it. As Soloway, Bonar and Ehrlich (1983) noted, some of the most bizarre and convoluted programs can be readily understood as attempts to make a "Get-Process" ordering work in spite of various obstacles. Empirical evidence supports this view: subjects overwhelmingly preferred "Get-Process" constructs to "Process-Get" ones (Soloway, Bonar & Ehrlich, 1983).

With evidence that suggests that students are taught to rely on the WHILE, and given evidence that the "Get-Process" ordering is the most natural, it is perfectly reasonable to find the "Get-Process" WHILE combination to be the preferred one; in fact, it would be somewhat surprising if it were not.

Why, then, is this most obvious combination the most fatal? To understand this, it is necessary to note that the WHILE construct itself, not properties of any task, is the source of the unnatural "Process-Get" ordering. The location of the test at loop entry creates the "Process-Get" ordering since it implies an initial "Get" prior to loop entry. This in turn dictates that "Process" steps occur after loop entry and before the within-loop "Get" to avoid losing the first value before it is processed.

Certain kinds of problems are natural occasions for the use of the WHILE, but it is not the problems that call for a "Process-Get" ordering. All such problems can be readily solved with the natural "Get-Process" ordering via a construct which supports test-and-exit in the middle of the loop. However, students programming in

Pascal do not have such a construct. They do have the WHILE and are taught to rely upon it. They seem to have mastered it quite well: we found insignificant number of failures to implement it legally. What seems invisible to them is that this construct implies an inherent "Process-Get" ordering. So, they implement the WHILE using the natural "Get-Process" ordering. As the data clearly indicates, this *order clash* between what the student naturally wants to do ("Get-Process") and what the WHILE construct invites ("Process-Get") reliably led to failure.

One major cause of failure is the incomplete instructional treatment given to the WHILE construct. Throughout the 27 texts examined were numerous example programs with WHILE loops and narrative descriptions of their execution. None of the examples evidenced a "Get-Process" WHILE. If "teaching by example" alone were sufficient, we would not see many "Get-Process" WHILE loops. The fact that we did indicates that annotated examples are not sufficient to overcome the natural propensity for a "Get-Process" ordering. Conventional WHILE loop treatments are inadequate: they fail to effectively acknowledge the danger the "Get-Process" WHILE construct presents to novices.

4.3. WHILE DEPENDENCE: ANOTHER CAUSE OF FAILURE

Another major cause of failure is the inaccurate instructional treatment of the iteration constructs and their effective usage. What we're facing here is not an error of one or two textbook authors; rather, it is a widespread myth of programming, i.e. "it doesn't really matter which loop construct is used." Our data indicates that for novices it matters very much. Subject performance improved dramatically when both the definition and the application of the WHILE loop were constrained. This flies in the face of the current treatment of the WHILE as flexible with respect to both construction and use.

Our subjects seem to have assimilated the current view of the WHILE quite well... with disastrous results (see Table 3). Problems 1 and 2 were designed explicitly to call for the FOR and REPEAT loops, respectively. Yet more subjects attempted to solve these problems by using the WHILE construct than the preferred ones: 80 WHILE loops compared to 64 preferred construct attempts. This reliance on the WHILE resulted in a dismal success rate: less than 24% overall for WHILE loop solutions to Problems 1 and 2. This is *less than half* the success rate associated with the preferred constructs.

These findings indicate that the WHILE should be repositioned relative to the FOR and REPEAT constructs, with each having comparable status with respect to application. Each is a "special case" construct, appropriate to certain specific situations; none is truly "powerful and generalizable" in terms of effective novice use. It is counterproductive to teach novices to have uncritical trust in this peculiar construct.

The current problematic positioning of the WHILE construct is both reflected in, and exacerbated by, another factor: *the absence of any effective basis for decision regarding loop construct selection.* None of the texts examined provided any explicit decision rule by which novices could discern which loop construct is preferred in a given situation. Furthermore, the informal decision rule implied in many texts, articulated herein as the descriptive behavior rule, was found to be not followable.

This is particularly troublesome in light of the widespread acknowledgement that

a major problem facing computer science is that of program engineering. Given the theme that we want software to be designed, not "hacked", we find a troublesome precedent being set at the earliest stages of programmer education: *students are given no reasonable basis for decision about how to best implement iteration.* In fact, it appears that students are given contradictory messages. On an explicit level they are told in various ways "*design your program first, then write the code*". On an implicit level, the real message seems to be "*use a WHILE loop and muck around with it until it works*".

For novices, it is not true that the choice of looping construct doesn't matter. Nor is it true that the WHILE is truly generalizable with respect to application by novices. If it were, we would not see such dismal performance associated with WHILE loop solutions to problems which invite FOR and REPEAT constructs. It is not sufficient to simply teach students how the various loop constructs work and to give them no guidance with respect to effective usage. Conventional instructional treatment of the Pascal iteration constructs is inaccurate in (a) implying that construct choice doesn't really matter and (b) teaching novices to rely on the WHILE construct regardless of situation.

5. Recommendations

Iteration is among the most basic and frequently applied concepts in the programmer's repertoire. Students need to have adequate procedures for deciding which iteration construct to use and how to best use it. At present, they do not. As a consequence, novices currently have *no choice* but to blur the distinction between design and implementation. If we take the basic principles of software engineering seriously, we should not be introducing students to programming in this way. We must find a better way.

This study's findings suggest specific modifications to the current treatment of the Pascal iteration constructs. These modifications affect both construct choice decisions and construct implementation decisions. In our study, these modifications provided immediate significant improvement in novice programming performance.

5.1. FOR INSTRUCTIONAL PURPOSES, THE CONVENTIONAL ORDER-BLIND PASCAL LOOP SCHEMA SHOULD BE REDEFINED TO BE ORDER-STRICT

The results of our study show that subject performance improved significantly due to instructions to implement looping constructs strictly with respect to ordering (as presented in Figure 2). In other words, subjects did better after being instructed that FOR and REPEAT loops should feature a "Get-Process" ordering, and that WHILE loops should evidence a "Process-Get" ordering.

Analysis of subject performance data indicates that it is particularly important to modify the treatment of the WHILE construct to acknowledge the danger it presents with respect to *order clash.* It appears that the main reason the order-strict schema has significant impact is because it causes WHILE loops to conform to the "Process-Get" ordering (as found in Figure 2). The "Get-Process" WHILE should be explicitly prohibited, and the order-strict schema achieves this.

We speculate that the conventional order-blind treatment of the Pascal iteration constructs effectively prevents them from being perceived as atomic units: does a

REPEAT loop imply a "Get-Process" or "Process-Get" ordering of loop body statements? The conventional order-blind schema means that we (and novices) simple don't know. By tying construct to Get/Process ordering (as in Figure 2), we allow novices to perceive each construct as an atomic unit. The effect is to move Get/Process ordering decisions from the lower level of trial-and-error loop construction to the higher level of loop construct choice.

5.2. ADEQUATE DECISION PROCEDURES SHOULD REPLACE UNQUALIFIED TRUST IN THE WHILE CONSTRUCT

In the absence of effective decision rules for construct choice, novices deploy the WHILE construct indiscriminately across all problems. The results are disastrous. This appears to be because the application of the WHILE construct to problems which invite the FOR or REPEAT constructs has the consequence of encouraging the use of the failure-prone "Get-Process" WHILE.

It appears that novices are implicitly taught to trust the WHILE construct regardless of situation. This invites failure and must be changed. The fact that it is logically possible to implement any kind of iterative solution via the WHILE *does not justify* teaching students that it should be treated as if it truly "generalizable"; in terms of effective *use* by novices, it is not.

5.3. ADEQUATE DECISION PROCEDURES IMPROVE PERFORMANCE; AT THE NOVICE LEVEL, WE NEED TO DEVELOP BETTER DECISION PROCEDURES

The results of our study show that subject performance improved most when the order-strict loop schema was combined with the constructive use rule. The constructive use rule works for three of the most common looping problems; we are working on amplifying this rule to handle the "*n* and a half loop" case.

The traditional descriptive behavior rule was simply not followable by the subjects; it had no impact whatsoever on subject decisions. This finding underlines the immediate need for practical decision rules which provide a *reasonable* basis for systematic program design decisions.

In summary, we found a significant improvement in novice programming performance by modifying the instructional treatment of Pascal. We find this particularly encouraging and exciting because it shifts our focus from things we can't realistically do much about (i.e. "change Pascal") to things that we, and computer science educators everywhere, can effect: improving computer science instruction by augmenting existing systems with new insights.

We invite and encourage educators/researchers to investigate related questions, including:

- How might the constructive use rule be amplified to cover the "*n* and a half" loop case?
- What other rules might serve as a more effective basis for novice program design decisions with respect to iteration?
- In what other aspects of programming does novice performance suffer for lack of effective decision procedures?
- In what other aspects of programming is conventional instructional treatment inadequate with respect to effective language use?

References

CONWAY, R., GRIES, D. & ZIMMERMAN, E. C. (1981). *A Primer on Pascal.* Cambridge, MA: Winthrop.

COOPER, D. & CLANCY, M. (1982). *Oh! Pascal!* New York: W. W. Norton.

DALE, N. (1983). *Introduction to Pascal and Structured Design.* Lexington, MA: D.C. Heath.

DIJKSTRA, E. (1973). Personal communication to D. Knuth, cited in KNUTH, D. (1974), Structured programming with go to statements. *Computing Surveys,* **6,** 261–303.

FINDLAY, W. & WATT, D. A. (1978). *Pascal, an Introduction to Methodical Practice.* Potomac, MD: Computer Science Press.

GEAR, C. W. (1983). *Programming in Pascal.* Chicago, IL: SRA Publishing.

GRAHAM, N. (1983). *Introduction to Pascal.* St Paul, MN: West.

GROGONO, P. (1978). *Programming in Pascal.* Reading, MA: Addison-Wesley.

HOLT, R. C. & HUME, J. N. P. (1980). *Programming in Standard Pascal.* Reston, VA: Reston.

JONES, J. (1986). *Pascal: Problem Solving and Programming Style.* New York: Harper Row.

KOFFMAN, E. (1986). *Problem Solving and Structured Programming in Pascal.* Reading, MA: Addison-Wesley.

LAMB, R. (1986). *Pascal: Structure and Style.* Menlo Park, CA: Benjamin Cummings.

LEDGARD, H. (1986). *Pascal with Excellence: Programming Proverbs.* Rochelle Park, NY: Hayden.

LEESTMA, S. & NYHOFF, L. (1986). *Pascal: Programming and Problem Solving.* New York: Macmillan.

NANCE, D. (1986). *Pascal: Understanding Programming and Problem Solving.* St Paul, MN: West.

NANNEY, T. R. (1985). *Computing and Problem Solving with Pascal.* Englewood Cliffs, NJ: Prentice Hall.

PETERS, J. (1986). *Problem Solving with Pascal: Programming, Methods, and Algorithms.* New York: Holt, Rinehart & Winston.

POLLACK, S. V. (1983). *Introducing Pascal.* New York: Holt, Rinehart & Winston.

PYLE, I. C. (1981). *The Ada Programming Language.* Englewood Cliffs, NJ: Prentice Hall.

RICHARDS, J. L. (1982). *Pascal.* New York: Academic Press.

SAVITCH, W. (1986). *Pascal, an Introduction to the Art and Science of Programming.* Menlo Park, CA: Benjamin, Cummings.

SCHNEIDER, G. M. & BRUELL, S. C. (1981). *Advanced Programming and Problem Solving with Pascal.* New York: John Wiley.

SCHNEIDER, G. M., WEINGART, S. W. & PERLMAN, D. M. (1982). *An Introduction to Programming and Problem Solving with Pascal.* New York: John Wiley.

SHACKELFORD, R. (1989). *The impact of loop schema feedback messages on looping strategy selection and program correctness.* Doctoral dissertation. Georgia Institute of Technology, Atlanta.

SKVARCIUS, R. (1984). *Problem Solving Using Pascal.* New York: PWS.

SMITH, P. (1986). *Introduction to Computers and Programming: Pascal.* Belmont, CA: Wadsworth.

SOLOWAY, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM* **29,** 850–858.

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., BONAR, J. & GREENSPAN, J. (1982). What do novices know about programming? In A. N. BADRE & B. SCHNEIDERMAN, Eds. *Directions in Human–Computer Interaction.* New York: Ablex.

SOLOWAY, E. & EHRLICH, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering,* **SE-10,** 1132–1139.

SPOHRER, J. & SOLOWAY, E. (1986). Novice mistakes: are the folk wisdoms correct? *Communications of the ACM,* **29,** 624–632.

WALKER, H. (1986). *Introduction to Computing and Computer Science with Pascal.* Boston, MA: Little, Brown, and Co.

WELLS, T. (1986). *A Structured Approach to Building Programs in Pascal.* New York: Yourden Press.

WELSH, J. & ELDER, J. (1979). *Introduction to Pascal.* Englewood Cliffs, NJ: Prentice Hall.

WETZEL, G. & BULGREN, W. (1986). *Pascal and Algorithms: an Introduction to Problem Solving.* Chicago, IL: SRA Publishing.