ED 406 211 SE 059 943

AUTHOR Madison, Sandra; Gifford, James
TITLE Parameter Passing: The Conceptions Novices Construct.
PUB DATE Mar 97
NOTE 30p.; Paper presented at the Annual Meeting of the American Educational Research Association (Chicago, IL, March 24-28, 1997).
PUB TYPE Reports - Research/Technical (143) -- Speeches/Conference Papers (150)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS *Computer Science; *Constructivism (Learning); Higher Education; *Misconceptions

ABSTRACT
This study explores the parameter-related misconceptions of two college students enrolled in a Pascal programming course. Findings indicate that both students appeared to conceive a direct procedure-to-procedure linkage, with the connection being made by identically named formal parameters. Both students were able, by making seemingly innocuous adjustments to formal parameter lists, to construct correctly functioning modular programs. They were also able to correctly answer parameter-related questions as long as the questions did not provoke a conflict with their fundamental misconceptions of the parameter process. As a result, the misconceptions were hidden from the instructor and perhaps from the students themselves. The results are discussed within a constructivist framework and implications for instruction are suggested. Contains 22 references. (Author/JRH)

Parameter Passing

The Conceptions Novices Construct

Sandra Madison, University of Wisconsin-Stevens Point

James Gifford, University of Wisconsin-Stevens Point

Paper presented at the Annual Meeting of the American Educational Research Association
Chicago, Illinois

March 1997

Parameter Passing

The Conceptions Novices Construct

Sandra Madison, University of Wisconsin-Stevens Point
James Gifford, University of Wisconsin-Stevens Point

Abstract

This study explored the parameter-related misconceptions of two college students enrolled in a Pascal programming course. Both students appeared to conceive a direct procedure-to-procedure linkage, with the connection being made by identically named formal parameters. Both students were able, by making apparently innocuous adjustments to formal parameter lists, to construct correctly functioning modular programs. They were also able to correctly answer parameter-related questions as long as the questions did not provoke a conflict with their fundamental misconception of the parameter process. As a result, the misconceptions were hidden from the instructor and perhaps from the students themselves. The paper discusses the results within the constructivist framework and suggests implications for instruction.

Background

Student interest in the computing sciences and information systems, after reaching a peak in the early 1980s, has declined dramatically. When coupled with increased industry demand for computing professionals and higher than average salaries, the decline is especially troublesome.

The literature repeatedly confirms that many students find college-level computer programming courses demanding, that they are often unsuccessful, and that attrition rates are high. Various authors offer reasons why students find programming courses difficult. Hence, it is not surprising that research reports of studies attempting to identify factors of computer programming success appear frequently. Although some studies have found positive correlations with computer programming ability, such studies generally are not very helpful to programming teachers. While background factors may assist advisors counseling students regarding the selection of programming classes, teachers cannot alter their students' age, gender, verbal ability, high school rank, or other background data. Programming instructors need research that informs the instructional and curricular decisions over which they have control. As Spohrer and Soloway (1986) suggested, "The more we know about what students know, the better we can teach them" (p. 624).

Understanding the programming language constructs is a crucial link in the cognitive chain of learning to program (Linn, 1985; Putnam, Sleeman, Baxter, & Kuspa, 1986). Thus, it is not surprising that researchers have investigated various language constructs, attempting to describe expert and novice conceptions of those constructs (Du Boulay, 1986; Kurland, Pea, Clement, & Mawby, 1986; Samurçay, 1985; Sleeman, Putnam, Baxter, & Kuspa, 1988). What is surprising is the virtual absence of attention to the construct of parameter (or argument) passing, the mechanism by which data are shared among the various program modules in a complex program (Fleury, 1991; Madison, 1995). The topic is pivotal since it is precisely the feature that gives modern programming languages their power (McIntyre, 1991). Pedagogical understanding of the topic is made even more important by the fact that many instructors and students attest to the fact that parameter passing is the most challenging concept covered in the introductory programming course (Cigas, 1995; Madison, 1995).

Purpose

This study was part of a larger study with the general purpose of describing novice programmers' understanding of the construct of simple parameter passing and the ultimate goal of improving the way in which the parameter concept is taught. The current study specifically addressed the following question: What misconceptions do novices possess regarding the concept of Pascal parameters? It provides part of the answer by pursuing an in-depth analysis of two students' naive conceptions.

Theoretical Perspective

The study assumes a constructivist perspective. It presumes that people construct knowledge through a "process of building intellectual structures that change and interact and combine" (Papert, 1988, p. 3) rather than being empty vessels to whom knowledge can be transmitted. Specifically, the theoretical framework for this study builds on the Linn and Songer (1991) model for conceptual change that was originally posed for students learning science. The three stages of the model include action knowledge, intuitive conceptions, and scientific principles. Action knowledge is the understanding learners bring to a learning situation, understanding that is based on personal experience. Intuitive conceptions are the conjectures learners make to explain events they observe as they attempt to make sense of the world they perceive. Intuitive conceptions, since they solely derived from unexamined experience are often incorrect, and might well be called "misconceptions." During the last stage of the model, learners acquire scientific principles, organizing their intuitive conceptions into principles consistent with those held by experts. Characteristic of the intuitive conception stage of the model, researchers studying novice programmers have found that, rather than resulting from slips in mechanics of program construction, most faulty answers arise from systematic application of knowledge the student already has. The answers make sense when interpreted in terms of the student's current understanding (Pea, Soloway, & Spohrer, 1987). Viewed in terms of the Linn and Songer model for conceptual change within a constructivist framework, teachers should not be surprised that students construct their own meanings and that they frequently harbor misconceptions in spite of instruction to the contrary.

Method

The current study was a theory-driven study drawing upon several methodologies traditionally employed in qualitative educational research. Consonant with recent research in education, this study focused on "detailed analyses of individual subjects rather than on aggregate behavior" (Eylon & Linn, 1988, p. 285). Data sources included observation of the classroom instruction, semi-structured interviews, protocol analysis, and document examination. This paper focuses on two of the students (with self-selected pseudonyms Jason and Steve) who participated in a larger exploratory study of novices learning the Pascal parameter process.The bulk of the data for the larger study was collected from interviews with students enrolled in one section of the introductory programming course at a regional midwestern university during a fall semester.

Technical note: The study included no tasks that involved nested procedures, nested procedure calls, or structured parameters. The interview tasks included no loops, selections or other program constructs that could be construed as confounding the data. Moreover, the scope rules were further simplified because the instructor availed himself of Turbo Pascal's facility to make the main module variables local rather than global.

Subjects

After a priori exclusion of students who had been previously exposed to the Pascal parameter process, participants were selected for the full study based on their score on the last six subtests of the Inventory of Piaget's Developmental Tasks (IPDT) (Furth, 1970). The IPDT has been shown to be a valid (by comparing scores on the written group test with scores obtained from classic Piaget interviews) and reliable (using the test-retest and split-half measures) indicator of the Piagetian view of cognitive development (Milakofsky & Patterson, 1979;

Cafolla, 1987-88). For purposes of this study, 24 was a perfect score, presumably suggesting a formal thinker.

In the several years preceding this study, more than 300 programming students at the university in question had completed the IPDT. Theory and classroom experience had already provided persuasive evidence that students with low scores struggle to acquire the same understandings of the abstract programming concepts that seem to come almost naturally for some students with high scores. Thus, it was reasonable to assume that students from the two groups might have differing conceptions of the parameter construct. While there did appear to be a relationship between IPDT scores and problem-solving styles, there seemed to be no indication that students with either a high or low IPDT score would be more prone to develop fundamental misconceptions. Jason's score of 22 was one of the highest; Steve's score of 17 one of the lowest. Steve had had some prior Pascal experience; Jason had not.

Results and Discussion

Jason and Steve both exhibited problem-solving behaviors that could be attributed to a fundamental misconception of the parameter process. Errors they made supported an interpretation suggesting a systematic application of a parameter process that involved a direct procedure-to-procedure communication link, with the connection being made through identical formal parameter names. Consistent with the literature on misconceptions, some oral and written responses that initially appeared capricious--and even bizarre--proved to be logical and appropriate choices when interpreted in terms of the learners' hypothesized misconceptions. Importantly, both students were able--despite their fundamental misconceptions--to repeat the textbook's and instructor's language, to correctly answer test questions, and to construct modular programs that produced the correct answer by making seemingly innocuous adjustments to parameter lists. Thus, the students' lack of understanding was concealed from the instructor and possibly from the students themselves.

Jason

In contrast to many novice programmers who experience difficulty with parameters, Jason had no difficulty with his value-variable parameter choices. He consistently chose correctly. However, the reason he gave--that variable parameters can be sent to other procedures--suggests that he did not make his choices for the right reason. As the following protocol suggests, he knew when variable parameters were needed; he just did not know where they went.

During a hand-construction task (see Appendix A for the task and Appendix B for Jason's completed program) in which the parameter name choice was constrained, Jason failed to choose appropriate parameter names. Jason approached the task by briefly examining the main module; then turned to the first procedure on the page. In his hand-written construction (which follows)

```
PROCEDURE CalculateVolume (     Len : Real;
                                Wid : Real;
                                Dep : Real;
                            VAR Volume: Real);
```

of the procedure heading line for CalculateVolume, Jason correctly named Len, Wid, and Dep "because of their use in the assignment statement `Volume := Len * Wid * Dep;`". He made them value parameters because, in his words, "they are only going to be used in this procedure." Volume he made a variable parameter, citing as his reason, "it has to be sent, available for the DisplayOutput procedure to use." Those familiar with Pascal will recognize that the code above would not compile. In all likelihood, Jason would have written a procedure call to match his procedure had that been an option. Next, Jason worked on DisplayOutput, the second procedure on the page. He constructed the following procedure heading line:

`PROCEDURE DisplayOutput(Volume: Real);` , giving the procedure a value parameter because "it didn't need to be sent anywhere." However, he named the parameter Volume rather than Answer, the only identifier used in the procedure's only executable statement

`WriteLn ('The volume of the solid is ', Answer:8:2);` because

> it has to come from the above procedure [CalculateVolume]. Volume was used as a variable parameter up there [Volume in CalculateVolume], so it has to be sent down to this procedure DisplayOutput.

One interpretation of Jason's observations is that he imagined a direct variable-to-value parameter connection between the two procedures, a connection that was established with identical parameter names.

Last, Jason examined the GetInput procedure, the third procedure on the page. The main module procedure calls--that were provided--and Jason's procedure heading line follow.

```
GetInput (Depth);
GetInput (Width);
GetInput (Length);

PROCEDURE GetInput (VAR Len :Real;
                    VAR Wid : Real;
                    VAR Dep : Real;)
```

He incorrectly gave the procedure three formal parameters--Len, Wid, and Dep--which he correctly identified as variable parameters "so they can be sent to the procedure to perform the calculation," the CalculateVolume procedure. A general question about the appearance of the GetInput parameter list did not provoke further attention to the number of parameters in the list.

One interpretation of Jason's formal parameter name choices in the GetInput and DisplayOutput procedures suggests that he was not sensitive to the fact that the scope (the parts of the program that may use a particular identifier) of a parameter name is limited to the procedure in which the formal declaration appears. It also suggests that he may not have realized the formal parameter list provides the declarations for the variables used in the statements of the procedure. Jason used identical actual and formal parameter names (meaning that every parameter name matched a main module variable name, effectively making the main module variable names global) in every task that allowed him to do so. Moreover, he commented that he always used the same names. Doing so may have camouflaged the fact that Jason did not understand the scope of the identifiers. If there were no other evidence, this would be a reasonable and sufficient explanation for Jason's choices.

Conversely, Jason may have understood the scope rules. If he understood the scope rules but did harbor the hypothesized misconception regarding the direct procedure-to-procedure, name-connected link, his conception of the parameter process would have clashed with his knowledge of the scope rules *if the various procedures used different formal parameter names to refer to the same memory location.* His consistent use of identical variable and parameter names throughout a program may have allowed him to reconcile his naive parameter conception with the scope rules. Moreover, the tactic would have allowed him to construct programs that compile, execute, and produce the correct answer.

Interpreted from the perspective of Jason's hypothesized name-controlled, direct-connection link, his parameter choices were appropriate and his reasoning cogent and coherent. CalculateVolume provided a Volume to DisplayOutput; thus DisplayOutput needed a Volume to receive it. CalculateVolume needed Len, Wid, and Dep; thus GetInput needed to supply them. It is noteworthy that Jason examined the procedures in the order of their physical appearance on the page, rather than the logical order of their execution. Apparently that order determined the name choices.

Consonant with the hypothesized misconception, Jason appeared to have expunged the role of the actual parameter list from the parameter process. He expressed confusion about the parameter that "had to be called at the end in the main program," adding:

> It's not so bad knowing the stuff that has to go to other procedures. I get mixed up on the actual parameters, which ones you have to call on at the end when you call it up, and I can't remember.

When asked what is provided in the actual parameter list, Jason replied, "That's what I'm not sure about. I really don't know that." Responding to a question about variable parameters, Jason said, "Those parameters are able to be passed on to other procedures. They can be moved to other procedures." The words Jason chose to describe his conception echoed his problem-solving behavior.

A second task (For the task, see Appendix C) added further evidence. After looking briefly at the program in the code tracing exercise, Jason summarized his intended approach:

> First I'm going to see what the procedure does. Now I'm looking at the main line. Both A and B are going to be stored as N, which is 3, so WriteLn (N) is going to be equal to 4. A and B are both going to be equal because they're both stored as the same value, which is given right here as 3 in the main line. The procedure adds 1 to N, which is 3.

Jason recognized that the changes to the variable parameters in the procedure Increment would be reflected in the calling module, the main module. A diagram he drew, illustrates that he knew there was some connection between the formal variable parameters A and B and the "global variable N."

$$\begin{tikzcd} A \arrow[d] & B \arrow[d] \\ N \arrow[r, phantom, "{=4}" below] & N \end{tikzcd}$$

The "global" variable N provided the value 3 for A and B. After the procedure added 1 to both, A and B replaced N. Jason was able to reconcile the different names--N versus A and B--because the main module only had one variable; it was the only choice. Jason continued, offering that a change to A or B changed N because A and B were variable parameters. Asked if he had any mental image of how that happened, Jason replied, "I don't know if I have an image. If I see a VAR, then I know it can be sent to other procedures." Referring to the diagram, Jason continued, "Basically I see these two numbers; N and N are in this order. That means A and B will be stored in that order. I just see two little arrows that go right there." To the question: "If they were value parameters, what would happen to changes made to A and B; that is, if the VAR were not there, what would N be?" Jason replied, "It would still be 4, I think. The VAR really doesn't . . . I don't think you need it in this procedure." Asked again: "If this were a value parameter, if A and B were value parameters, what would N be when this WriteLn (N) is executed?" Jason again answered, " It would still be 4." The question: "If it would still be 4, how do value and VAR parameters differ?" evoked the following response:

> If you have more than one procedure, say three, four procedures . . . if A and B would have to be sent from this procedure to a different procedure, then you need VAR, but this program is not big enough to need a VAR.

Jason's response was congruous with his other statements. Interpreted from the hypothesized name-controlled, direct procedure-to-procedure link, the parameters in the first procedure to be executed (the Increment parameters A and B) obtained their values from the "global" main module variable N. As the program had no other procedures to which the value needed to be sent, variable parameters were unnecessary.

Jason repeated the words of the text and lecture, he wrote correctly-functioning modular programs as long as he could choose matching parameter names, and he correctly answered multiple choice test questions related to the number of parameters in corresponding actual and formal parameter lists and other facts. As long as the task did not conflict with his misconception, he succeeded. One might hypothesize that when Jason's conception of the parameter process clashed with rules he had memorized but whose purpose he did not understand, his naive conception prevailed.

Steve

Steve could fluently recite the rules for value and variable parameter actions; he several times repeated that value parameters only send information to the procedure while variable parameters can also send it back to the main module. Steve, however, acknowledged that he sometimes used variable parameters "just to be on the safe side," implying that he was not always certain when value and variable parameters were appropriate. Mentioning several times that it was important to do so, Steve selected different actual and formal parameter names as he completed the interview tasks. In contrast, the original programs he wrote for the class used identical actual and formal parameter names throughout.

Just as the hand construction task in which the parameter names were constrained (See Appendix A) revealed Jason's misconception, so did it Steve's. Steve first attempted to construct the formal parameter list for GetInput. He mused, "I'm pondering whether these are local variables. I just don't understand. I'm confused by the three separate calls of the same procedure." He recognized that the input procedure needed a variable parameter "because it has to send it to `CalculateVolume (Length, Width, Depth, Volume)`. There's one thing, I know," he continued, "I know it reads them in the same order as they're entered in here because it needs to go to CalculateVolume." Steve's reference to the appearance of the two procedures' parameter lists (that the parameters needed to be in the same order) offered a clue that he imagined a direct procedure-to-procedure link.

As the GetInput procedure seemed to be confusing Steve, the interviewer suggested looking at the other procedures first. Doing so may have been a mistake. Because Steve was examining the procedures in the logical order of execution, the outcome of his efforts on the task may have been different had he continued with that approach.

Steve turned to the parameter list of CalculateVolume, the first procedure listed on the page. The procedure heading line (which follows)

```
PROCEDURE CalculateVolume (     Num1 : Real;
                                Num2 : Real;
                                Num3 : Real;
                            VAR Num4 : Real);
```

he constructed contained the correct number of formal parameters, and the value and variable choices were correct. However, Steve selected the parameter names Num1, Num2, Num3, and Num4, none of which matched the names used in the procedure's only executable statement:

```
Volume := Len * Wid * Dep
```

Steve offered that he had not chosen the names for any particular reason, that "the actual parameter list just needs something to match up with." Num4, he offered, needed to be a variable parameter because "it has to get passed between procedures to DisplayOutput after it's calculated." His phrase "passed between procedures" again suggested the direct link.

Steve next constructed the DisplayOutput parameter list (which follows).

```
PROCEDURE DisplayOutput(     Answer: Real)
```

He correctly named the single value parameter Answer "because the WriteLn calls it up as Answer." However, he also changed the Num4 parameter in the CalculateVolume procedure to Answer "because Volume is associated with Answer, but you cannot have the same name in the main line and in the procedure." The revised procedure heading line follows.

```
PROCEDURE CalculateVolume (     Num1 : Real;
                                Num2 : Real;
                                Num3 : Real;
                            VAR Answer: Real);
```

Steve allowed that the program would compile if the actual and formal parameter names were the same, but said it would confuse the reader of the program.

As Steve had correctly chosen the identifier Answer because of its use in the DisplayOutput procedure's WriteLn, he apparently knew that variable names must be consistent within a procedure. However, the instructor had strongly stressed the importance of using different names for the actual and formal parameters, an admonition that apparently impressed Steve powerfully. As written, the CalculateVolume procedure demanded that the formal parameter be named Volume, a requirement that conflicted with the instructor's counsel regarding parameter names. One could conjecture that when Steve encountered a conflict between the instructor's caution and the consistency requirement for variable names within a procedure, the admonition prevailed. However, other than possibly focusing Steve's attention on choosing formal parameter names that differ from the actual parameter names, and away from the need for consistency within the procedure, the explanation does not account for his choices of Num1, Num2, and Num3 in the CalculateVolume procedure. Alternatively, it may be that the added level of abstraction imposed by the requirement of different actual and formal parameter names hampered Steve's construction efforts. As his correctly done classwork had used identical names throughout, the interpretation seems plausible. The combination of evidence, however, will suggest a more cogent interpretation for Steve's construction.

The other procedure heading lines completed, Steve returned to GetInput. He once more alleged that the parameters needed to be VARs because "they have to go to other procedures." After an extended pause, Steve said:

> I don't know why it wouldn't work the same. If it were just regular, as if they were all in one line [In contrast to the input procedure in this program that was called several times, most input procedures Steve had seen obtained all the necessary input for the program during a single call.] But it's a new call each time. This is basically a part that I am having trouble dealing with.

It appears that Steve began by imagining a correctly constructed GetInput parameter list (which follows).

```
PROCEDURE GetInput (VAR Dimension: Real);
```

He admitted that this portion of the task was "kind of confusing," and correctly speculated that when the procedure GetInput (Depth) is called, it will "go through and run the procedure and associate Dimension with Depth. Then he questioned:

> But then when you call GetInput (Width), is it going to know to run the procedure and instead of putting it in Depth again, will it put it in the next one?

Steve had earlier commented about not understanding how the procedure header and the procedure call should be written and not understanding the process on the "syntactical level." His statements could mean that he did not understand the connection between an actual parameter and a formal variable parameter. He knew that a variable parameter changes a value, but it may be that he was unsure where the change happened. For example, Steve had earlier said: "A value parameter passes only from the main line to the procedure, and a variable parameter can be passed between procedures and back and forth between the main line and procedures." Speaking of changes to variable parameters he had also offered, "Then it's changed completely. If you pass a variable from a procedure to another procedure and that procedure changes it, then that is the value that's stored in memory." His statements echoed those reported by Fleury (1991).

Steve constructed the GetInput to match what he was more accustomed to seeing--one call to a procedure that secured all the inputs the program needed--naming the variable parameters Num1, Num2, and Num3. The provided procedure calls and the GetInput procedure heading line that Steve constructed to match the calls follow.

```
GetInput (Depth);
GetInput (Width);
GetInput (Length);

PROCEDURE GetInput (VAR Num1: Real;
                    VAR Num2: Real;
                    VAR Num3: Real;)
```

As Steve constructed numerous correctly-functioning procedures over the course of the semester, one must assume he knew that the number of actual and formal parameters must match. However, the conflict provoked by the seemingly novel input procedure caused Steve to abandon or distort the procedure-line construction rules--rules whose purpose he perhaps did not understand--to allow him to build a procedure heading line consistent with his understanding.

Next, recognizing that the GetInput procedure needed a declaration to match the `ReadLn (Dimension)` statement, Steve added Dimension as a local variable. Referring to the GetInput procedure, he claimed, "That's the only one that is going to use the Dimension in that form because it's not used anywhere throughout, so it's not going to be put in the VAR parameter. It's a local variable." It appears Steve was imagining a direct procedure-to-procedure connection between the parameters, a connection that was established through identical parameter names. As no other procedure had a parameter named Dimension, GetInput did not need one, either. (See Appendix D for Steve's completed program.)

Steve next completed a modularization task, converting a simple one module program into one composed of several cohesive procedures. (See Appendix E.) Although Steve's modularized version of the program consistently used different actual and formal parameter names, the formal parameter names were identical in all procedures, just as they were (inappropriately) in the preceding task. If Steve's hypothesized intuitive conception of variable parameters is accurate, then he found a way to reconcile his conception (that information is conveyed between procedures via a procedure-to-procedure connection established by identical formal parameter names) with the instructor's requirement that actual and formal parameter names differ.

At the conclusion of his interview, Steve once more explained the difference between value and variable parameters, saying, "A value parameter is passed only from the main line to the procedure and not back. Variable parameters pass back and forth between procedures and the main line, depending on how many procedures and whatnot. If it's a variable parameter, it can send it somewhere else." The phrases "depending on how many procedures and whatnot" and "send it somewhere else" were consistent with the hypothesized procedure-to-procedure linkage.

Many of Steve's problem-solving decisions and behaviors approximated Jason's. As Steve constructed numerous correctly functioning procedures over the course of the semester, he presumably knew the parameter rules. However, when his choices were constrained, he--like Jason--did not consistently choose names compatible with the procedure statements or construct formal parameter lists with the correct number of parameters. He chose instead formal parameter names that were identical in all procedures that referred to the same data element. Speaking of changes to variable parameters, Steve said: "If you pass a variable from a procedure to another procedure and that procedure changes it, then that is the value that's stored in memory" and "Once it's stored in the computer memory, it can be used throughout." It seemed that Steve was imagining that changing a VARed parameter made it global (known "throughout"), an accurate depiction if all procedures employ identical formal parameter names when referring to the same memory location--as Steve's programs did.

Implications

Steve and Jason exhibited problem-solving behaviors and verbalizations that could be attributed to a fundamental misconception of the parameter process. Each made errors that could be interpreted as systematic applications of a parameter process that involved a direct procedure-to-procedure communication. This finding has several important implications.

First, Jason was a formal thinker and consistently exhibited problem-solving behaviors that would be described as analytic (scrutinized the abstract analysis tasks without imbedding the code in a familiar context, did not need to stay in close contact with the details of a problem situation, discerned the general purpose of a program before examining the details, attended to the logical representation of programs, showed little evidence of relying on trial-and-error as a problem-solving approach, and alluded to control when describing human-machine interactions.). As demonstrated by his score on the IPDT, Steve was a concrete thinker. Moreover, he consistently employed a problem-solving style that appeared to be concrete (placed the abstract analysis tasks in context before analyzing them, attempted to remain in contact with a problem's details, often examined the details of a problem at length before discovering its purpose, occasionally seemed bound by a program's physical representation, regularly employed trial-and-error as problem-solving approach, and described interactions with the computer as a partnership). Thus it is not possible to conclude, as might have been expected, that concrete, contextual thinkers are more prone to misconception than formal, analytical thinkers. Tobias (1990) advised against assuming "too narrow a vision of what kinds of attributes, behaviors, and lifestyles the 'true' scientist displays" (p. 14). The evidence from this study suggests that her counsel applies to assumptions about novice programmers as well.

Jason's and Steve's choices made sense when interpreted in terms of the hypothesized misconception. Thus, the finding suggests that programming teachers refrain from interpreting program errors as simply carelessness or lack of attention or lack of concern on the student's part.

It is also important to note that, during the interviews, both participants verbalized their inability to understand. Although the words varied, the common message was, "I do not understand." This fact suggests that students can provide helpful information about their understandings and misunderstandings if they are provided an opportunity. In recent years, student journals and other classroom assessment techniques have gained great popularity; incorporating their use in an introductory programming class could provide novices with the opportunity to say, "I do not understand."

Teachers might also be alerted to student difficulties by using a face-to-face grading scheme (Cooper, 1985). Obviously, an instructor teaching multiple classes of 30 to 35 (or more) programming students cannot engage in face-to-face grading with every student. However, journals or other classroom assessment techniques could identify students who would profit most from the interchange. The practice seems to offer many potential benefits. Students who explain a program's function, and justify design decisions, could obtain the same benefit that interview participants did: the opportunity to elaborate their conceptions. Moreover, one could hypothesize that students would be less inclined to submit a program they did not fully understand if they knew they might be required to explain the program's functioning to their instructor. Finally, the scheme ensures that students receive immediate feedback on their work.

Both Jason and Steve could write original programs that produced the correct answers by making apparently innocuous adjustments to parameter names and value-variable choices, thus allowing them to construct programs that produced the correct results despite fundamental misunderstandings of the parameter construct. If understanding of the programming language constructs is a legitimate and important goal of programming instruction, the findings suggest that instructors of introductory programming courses must structure opportunities whereby learners construct criteria for programming success that are more comprehensive than merely "finding the right answer." The instructor who relies heavily on original program construction, particularly computer-assisted program construction, has no assurance that students understand the fundamental language constructs.

Future Directions

The study was subject to all the limitations inherent in a qualitative study (Merriam, 1988). Moreover, of the instruments and protocols employed, only the IPDT had undergone the rigor of validity and reliability testing (Milakofsky & Patterson, 1979). With the exception of the analysis task described in Appendix C that was excerpted from the Advanced Placement Course Description (College Board, 1993, p. 41), the tasks utilized in the exploratory study were the invention of the researcher. They reflect one instructor's view of skills and understandings that might be important to assess when studying novice Pascal programmers learning the parameter passing construct.

Owing to the exploratory nature of the study, some tasks may not have probed participants' understanding as deeply as possible. Given, for example, the unexpected emergence of the direct procedure-to-procedure link misconceptions, tasks that incorporated only one procedure likely yielded less information regarding the novices' intuitive conceptions than would tasks with multiple procedures. Moreover, no tasks included variables superfluous to the parameter process; the likelihood of correct guessing was thereby increased. Additionally, no task required the programmer to construct actual parameter lists to match existing formal parameter lists. Attention to these limitations provides guidance for future researchers.

Educational Significance

An introductory programming course is traditionally taught as a skills development course. Programmers are, after all, ultimately judged by the skill with which they construct programs. Accordingly, traditonal instruction has been designed to develop skills, perhaps at the expense of understanding. The course in which the participants were enrolled was no exception. The results of this study suggest that educators may wish to rethink the goals, assessment, and methods of instruction of the parameter topic in an introductory computer programming course. Doing so may enable some students to construct an understanding that moves beyond the stage of intuitive conception to the stage of principled understanding of the parameter process.

References

Cafolla, R. (1987-1988). Piagetian formal operations and other cognitive correlates of achievement in computer programming. *Journal of Educational Technology Systems, 16,* 45-55.

Ciga, J. F. (1995). Proper packaging promotes parameter passign proficiency. *ACM SIGPLAN Notices, 30,* (4), 80.

College Board. (1993). *Advanced placement course description: Computer science.* USA: College Entrance Examination Board.

Cooper, D. (1985). *Teaching introductory programming.* New York: Norton.

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research, 2,* 57-73.

Eylon, B. & Linn, M. (1988). Learning and instruction: An examination of four research perspectives in science education. *Review of Educational Research, 58,* 251-301.

Fleury, A. (1991). Parameter passing: The rules the students construct. *SIGCSE Bulletin, 23* (1), 283-286.

Furth, H. (1970). An inventory of Piaget's developmental tasks. Center for Research in Thinking and Language, Department of Psychology, Catholic University: Washington, D. C.

Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (1986). A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research, 2,* 429-458.

Linn, M. C. (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher, 14* (5), 14-16, 25-29.

Linn, M. C. & Songer, N. B. (1991). Cognitive and conceptual change during adolescence. *American Journal of Education, 99,* 379-417.

Madison, S. K. (1996). *A Study of College Students' Construct of Parameter Passing: Implications for Instruction.* ERIC ED 390 378.

McIntyre, P. (1991). *Teaching structured programming in the secondary schools.* Malabar, FL: Krieger.

Merriam, S. H. (1988). *Case study research in education: A qualitative approach.* San Francisco: Jossey-Bass.

Milakofsky, L. & Patterson, H. O. (1979). Chemical education and Piaget. *Journal of Chemical Education, 56*, 87-90.

Papert, S. (1988). The conservation of Piaget: The computer as grist to the constructivist mill. In G. Forman & P. Pufall (Eds.), *Constructivism in the computer age* (pp. 3-13). Hillsdale, NJ: Lawrence Erlbaum.

Pea, R. D., Soloway, E., & Spohrer, J. C. (1987). The buggy path to the development of programming expertise. *Focus on Learning Problems in Mathematics, 9*(1), 5-30.

Putnam, R. T., Sleeman, D., Baxter, J. A., & Kuspa, L. K. (1986). A summary of misconceptions of high school basic programmers. *Journal of Educational Computing Research, 2*, 459-472.

Samurçay, R. (1985). Learning programming: An analysis of looping strategies used by beginning students. *For the Learning of Mathematics, 5*, 37-43.

Sleeman, D., Putnam, R. T., Baxter, J., & Kuspa, L. (1988). An introductory Pascal class: A case study of errors. *Journal of Educational Computing Research, 4*, 5-23.

Spohrer, J. C. & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM, 29*, 624-632.

Tobias, S. (1990). They're not dumb. They're different. "A new tier of talent" for science. *Change*, 11-29.

This paper is based on portions of Sandra Madison's dissertation, completed at the University of Wisconsin-Milwaukee under the supervision of Henry S. Kepner, Jr.

Appendix A

Complete the missing procedure heading lines so that this program will compile and do the job implied by its name. Remember that the formula for calculating the volume of a rectangular solid is: volume = length x width x depth. As you complete each procedure heading line, explain as completely as you can why you constructed it as you did.

```
PROGRAM CalculateVolumeOfARectangularSolid;
...............................................
PROCEDURE


BEGIN
  Volume := Len * Wid * Dep
END;
.....................................................................
PROCEDURE


BEGIN
  WriteLn ('The volume of the solid is ', Answer : 8 : 2)
END;
.....................................................................
PROCEDURE


BEGIN
  Write ('Please enter a dimension ==> ');
  ReadLn (Dimension)
END;
{main}.................................
VAR
  Volume  : Real;
  Length,
  Width,
  Depth:    Real;

BEGIN
  GetInput (Depth);
  GetInput (Width);
  GetInput (Length);
  CalculateVolume (Length, Width, Depth, Volume);
  DisplayOutput (Volume)
END.
```

Appendix B
Jason's Hand Construction

Complete the missing procedure heading lines so that this program will compile and do the job implied by its name. Remember that the formula for calculating the volume of a rectangular solid is: volume = length x width x depth. As you complete each procedure heading line, explain as completely as you can why you constructed it as you did.

```
PROGRAM CalculateVolumeOfARectangularSolid;
..............................................
PROCEDURE CalculateVolume (Len, Wid, Dep: Real; VAR Volume: Real)


BEGIN
  Volume := Len * Wid * Dep
END;
....................................................................
PROCEDURE DisplayOutput (Volume: Real)


BEGIN
  WriteLn ('The volume of the solid is ', Answer : 8 : 2)
END;
....................................................................
PROCEDURE GetInput (Var Dep, Wid, Len : Real)


BEGIN
  Write ('Please enter a dimension ==> ');
  ReadLn (Dimension)
END;
{main}...................................
VAR
  Volume  : Real;
  Length,
  Width,
  Depth:    Real;

BEGIN
  GetInput (Depth);
  GetInput (Width);
  GetInput (Length);
  CalculateVolume (Length, Width, Depth, Volume);
  DisplayOutput (Volume)
END.
```

Appendix C
Program Analysis Task

```
PROGRAM ABC;

PROCEDURE Increment (VAR A,
                     B: Integer);
BEGIN
  A := A + 1;
  B := B + 1
END;

VAR
  N: Integer;
BEGIN
  N := 3;
  Increment (N, N);
  WriteLn (N)
END.
```

Appendix D
Steve's Hand Construction

Complete the missing procedure heading lines so that this program will compile and do the job implied by its name. Remember that the formula for calculating the volume of a rectangular solid is: volume = length x width x depth. As you complete each procedure heading line, explain as completely as you can why you constructed it as you did.

```
PROGRAM CalculateVolumeOfARectangularSolid;
...............................................
PROCEDURE CalculateVolume (    Num1: Real;
                               Num2: Real;
                               Num3: Real;
                           Var Answer: Real);


BEGIN
  Volume := Len * Wid * Dep
END;
...................................................................
PROCEDURE DisplayOutput (Answer: Real);


BEGIN
  WriteLn ('The volume of the solid is ', Answer : 8 : 2)
END;
...................................................................
PROCEDURE GetInput (Var Num1: Real;
                    Var Num2: Real;
                    Var Num3: Real);
Var
  Dimension: Real;


BEGIN
  Write ('Please enter a dimension ==> ');
  ReadLn (Dimension)
END;
{main}.................................
VAR
  Volume  : Real;
  Length,
  Width,
  Depth:    Real;

BEGIN
  GetInput (Depth);
  GetInput (Width);
  GetInput (Length);
  CalculateVolume (Length, Width, Depth, Volume);
  DisplayOutput (Volume)
END.
```

Appendix E
Computer-Assisted Modularization Task
Initial Program

```
PROGRAM PayrollPreparation;
USES   Crt;

CONST
  InputPrompt2 = 'Please enter hours worked (1/2 hour increments) ==> ';
  InputPrompt3 = 'Please enter payrate (2 decimal places) ==> ';

{mainline (boss) variables}
VAR
  PayRate,
  HoursWorked,
  GrossPay  : Real;

BEGIN
  ClrScr;
  Writeln ('This program computes the gross pay for one employee');
  Writeln ('Overtime is not taken into consideration');
  Writeln;
  Write (InputPrompt2);
  Readln (HoursWorked);
  Write (InputPrompt3);
  Readln (PayRate);
  GrossPay := HoursWorked * PayRate;
  Writeln;
  Writeln ('Hours Worked  ': 12, 'Rate   ':10, 'Gross Pay  ':10);
  Writeln (HoursWorked:12:2, PayRate:10:2, GrossPay:10:2);
END.
```

Appendix E continued
Steve's Completed Modularized Program

```
PROGRAM PayrollPreparation;
USES   Crt;

PROCEDURE WelcomeUser;
  BEGIN
   WriteLn ('This program computes the gross pay for one employee');
   WriteLn ('Overtime is not taken into consideration');
   WriteLn;
  END;                  {END WELCOMEUSER}

PROCEDURE GetInput (VAR Hours : Real;
                    VAR Pay : Real);
   BEGIN
     Write ('Please enter hours worked in half hour increments: ');
     ReadLn (Hours);
     Write ('Please enter payrate (out to 2 decimal places): ');
     ReadLn (Pay);
   END;                 {END GETINPUT}

PROCEDURE CalculateWage (VAR Hours : Real;
                         VAR Pay : Real;
                         VAR Wage : Real);
   BEGIN
     Wage := Hours * Pay;
     WriteLn;
   END;                   {END CALCULATEWAGE}

PROCEDURE Display (VAR Hours : Real;
                   VAR Pay : Real;
                   VAR Wage : Real);
   BEGIN
     WriteLn ('Hours Worked ': 12,'Rate ':10,'GrossPay ':10);
     WriteLn (Hours:12:2, Pay:10:2, Wage:10:2);
   END;                  {ENDDISPLAY}

{mainline (boss) variables}
VAR
  PayRate,
  HoursWorked,
  GrossPay  : Real;

BEGIN
  ClrScr;
  WelcomeUser;
  GetInput (HoursWorked, Payrate);
  CalculateWage (HoursWorked, Payrate, GrossPay);
  Display (HoursWorked, Payrate, GrossPay);
END.
```

AREA 1997

SE059943

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement (OERI)
Educational Resources Information Center (ERIC)

ERIC

REPRODUCTION RELEASE
(Specific Document)

I. DOCUMENT IDENTIFICATION:

Title: Parameter Passing: The Conceptions Novices Construct	
Author(s): Sandra Madison and James Gifford	
Corporate Source:	Publication Date:

II. REPRODUCTION RELEASE:

In order to disseminate as widely as possible timely and significant materials of interest to the educational community, documents announced in the monthly abstract journal of the ERIC system, *Resources in Education* (RIE), are usually made available to users in microfiche, reproduced paper copy, and electronic/optical media, and sold through the ERIC Document Reproduction Service (EDRS) or other ERIC vendors. Credit is given to the source of each document, and, if reproduction release is granted, one of the following notices is affixed to the document.

If permission is granted to reproduce the identified document, please CHECK ONE of the following options and sign the release below.

[✓] Sample sticker to be affixed to document

Check here
Permitting microfiche (4"x 6" film), paper copy, electronic, and optical media reproduction

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
Sample
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

Level 1

Sample sticker to be affixed to document [ ]

or here
Permitting reproduction in other than paper copy.

"PERMISSION TO REPRODUCE THIS MATERIAL IN OTHER THAN PAPER COPY HAS BEEN GRANTED BY
Sample
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

Level 2

Sign Here, Please

Documents will be processed as indicated provided reproduction quality permits. If permission to reproduce is granted, but neither box is checked, documents will be processed at Level 1.

"I hereby grant to the Educational Resources Information Center (ERIC) nonexclusive permission to reproduce this document as indicated above. Reproduction from the ERIC microfiche or electronic/optical media by persons other than ERIC employees and its system contractors requires permission from the copyright holder. Exception is made for non-profit reproduction by libraries and other service agencies to satisfy information needs of educators in response to discrete inquiries."

Signature: Sandra K. Madison	Position: Associate Professor
Printed Name: Sandra K. Madison	Organization: University of Wisconsin-Stevens Point
Address: Dept. of Math + Computing UW-Stevens Point Stevens Point, WI 54481	Telephone Number: (715) 346-4612
	Date: 3/24/97

OVER