

PETTT TECHNICAL REPORT

PETTT-01-FA-01



Computer Science Education Research on Programming: What We Know and How We Know It

Lori Postner
August 2001

To cite this document, use the format:

Turns, J. and Wagner, T. (2001). Continuing Medical Education: Observations of CME Course on Feb. 8 and 9, 2001. Technical Report PETTT-01-PT-01. Program for Educational Transformation Through Technology, University of Washington, Seattle, Washington.

Notice: PETTT has made its working papers, reports, and other documents available for your free use, but only if you (a) reference us in your bibliography and (b) tell us about it by emailing us at pettt@u.washington.edu.

Mental Models

Several researchers have investigated beginners' mental models of programs. In most, a mental model is defined as the way the programmer understands what the computer does as it executes commands. A variety of methods have been used to learn about novices' mental models including verbal protocol and interviews.

Bayman and Mayer (1983) stated that two things happen as novices learn to program. First, beginning programmers acquire new information about the programming language and the rules that govern it. Second, they create a mental model of what happens inside the computer as the program statements get executed. Their study showed that often students create incorrect mental models although they can write proper programming statements. Bayman and Mayer looked at nine statements in BASIC and the most common misconceptions that arose within the subjects' mental models.

Canas, Bajo, and Gonzalvo (1994) also investigated students' mental models of programs. They believed that students who have help creating a mental model would have a better understanding of the program. In order to help students develop a mental model they provided students with a tracing mechanism that demonstrated how the computer worked as the code is executed. They found that students who used the tracing program had different mental models than those who did not. The students who used the trace mechanism organized the concepts by semantic aspects of the programming language whereas those who did not use the tracing organized by syntax.

Misconceptions/Bugs

In addition to literature on mental models, a number of researchers have looked at the types of mistakes students make as they program. In some cases they consider such mistakes misconceptions. In order to understand students' difficulties with programming, researchers have looked at students' programs and interviewed students.

Du Boulay (1986) categorized five areas that novices find difficult in learning to program:

1. orientation (what programming is)
2. notational machine (understanding the general properties of the computer)
3. notation (syntax and underlying semantics)
4. structures (various programming constructs)
5. pragmatics (specifying, developing, testing and debugging a program).

He continued to categorize the types of mistakes made by novice programmers into three groups:

1. misapplication of analogy
2. overgeneralizations
3. interactions.

A misapplication of analogy would entail taking an analogy too far. For example, many instructors talk about a variable as a 'box'. Using this analogy may cause students to believe that a variable can hold more than one value since a box can hold more than one object. When a novice programmer overgeneralizes he/she often makes general assumptions about a program's syntax which are inappropriate. Finally, when a student makes interaction mistakes he/she will use program sub-parts improperly. In this paper, Du Boulay shows a variety of mistakes that arise from students using their knowledge of English in programming.

Spohrer and Soloway (1986) found that the semantics of language constructs may not be the most common source of program bugs. In their study they looked at the types of bugs students create in syntactically correct programs. By coming up with plausible explanations for the various bugs they concluded that most bugs are not due to the students' misunderstanding of the

semantics of a programming construct but rather due to other problems. Their findings suggest that "Simply making the semantics of constructs clearer will not address many of the problems novices are having" (p. 191).

Perkins and Simmons (1988) looked at types of misconceptions across science, math and computer programming. They categorized four levels of knowledge that lead to misunderstandings:

1. content frame
2. problem-solving frame
3. epistemic frame
4. inquiry frame.

The content frame deals with the concepts most central to the domain, for example variables, expressions, assignment statements, loops, etc. This frame has the following types of misunderstandings associated with it.

- Naive, underdifferentiated, and malprioritized concepts.
- Difficulties in accessing knowledge.
- Problems of garbled knowledge.

The problem-solving frame deals with the ability to break down a problem into subparts in order to make the problem manageable. The following are examples:

- Trial and error.
- Preservation and quitting.
- Proceeding on a guess.
- Stock responses.
- Equation cranking.

The epistemic frame looks at the concepts and constraints of a domain thereby challenging the knowledge within that domain. Weaknesses in this frame are exemplified in the following:

- Intuitions mask contrary observation.
- Intuitions have priority over internal coherence.
- The grounding of the domain's rules is neglected.
 - Confirmation bias.

Finally, the inquiry frame focuses on extending the theory or challenging it. This is typically found in experts. Examples are:

- No problem finding.
- Academic applications only.
- No venturing.

Conceptual Framework

McGill and Volet (1997) developed a conceptual framework for analyzing computer programs from the literature on computer science education and cognition psychology. The combined three types of programming knowledge (syntactic, conceptual, and strategic) with three types of cognitive knowledge (declarative, procedural, and conditional) to form a five part framework (declarative-syntactic, declarative-conceptual, procedural-syntactic, procedural-conceptual, strategic/conditional knowledge). Within this framework *declarative-syntactic knowledge* is related to the syntax of a given language. For example, knowing the difference in C++ between a single equal sign, used for assignment, and a double equal sign (`==`), used to represent logical equivalence. *Declarative-conceptual knowledge* is the ability to understand and explain what happens as a program runs. It is the ability to demonstrate that one understands the semantics of the actions that are executed. For example, tracing through a code segment and determining it's output. Both declarative components have no bearing on the students' ability to apply this knowledge to writing statements. However, *procedural-syntactic knowledge* is the ability to apply one's understanding of syntax to writing statements in a specified language. For example, being able to write a syntactically correct for loop in C++. *Procedural-conceptual knowledge* is

using one's understanding of the semantics of code to write programs. For example, the ability to write a function to average three numbers. The final component, *strategic/conditional knowledge* is the ability to take one's comprehension of syntax and semantics to "design, code, and test a program to solve a novel problem" (p. 284). These categories can be used to define various ways students understand concepts presented to them in their introductory programming class. This paper describes several situations and the type of knowledge needed. The following table contains some of the examples provided in the paper (McGill & Volet, 1997, pp. 289-290).

Question	Types of knowledge
Why in the development of a program do we use program variables with meaningful names?	Declarative-conceptual
Explain the meaning of formal and actual parameters.	Declarative-syntactic (50%) Declarative-conceptual (50%)
How does a program activate a function?	Declarative-syntactic
For the following situation specify the parameters.	Procedural-conceptual
Write code for the procedure or function to return the larger or two numbers.	Procedural-syntactic (50%) Procedural-conceptual (50%)

Variables

Within researchers attempt to define mental models and describe misconceptions they have touched upon different aspects of a variable. This section discusses the different issues related with understand a variable in a variety of contexts.

Du Boulay (1986) identified misconceptions about variables based upon the analogies used in class. For example, the box or drawer with a label on it may lead students to believe that a variable can hold more than one value at a time. Another analogy he mentioned is a variable as a slate where values are written. Once again, students may arrive at a misunderstanding of a variable from this analogy. They may not realize that the existing value gets overwritten. Instead, they may think of a variable as a list that holds all values that have been assigned to that variable and can be retrieved. Another issue Du Boulay raised about students misconceptions about variables is that assigning one variable to another, for example $x = y$; may be viewed as linking the second variable to the first. Therefore a change in x results in a change in y . Du Boulay talks about novice users misunderstanding of temporal scope of variables. Students may not understand that a value stays in the variable until it is "explicitly changed or the contents of memory are erased or the machine is switched off" (p. 291). Finally, Du Boulay states "A common mistake, when using a variable to keep a running total, is to forget to initialize the total to zero. This omission is reasonable when following the box analogy. After all, if one has not put anything into a box, it's empty, which is sort of like zero" (291-292).

McCoy and Burton (1988) stated that mathematical ability is important to beginning programmers. They found that understanding mathematical variables, an important concept for novice programmers, may be able to predict success in programming courses. In a subsequent paper McCoy (1990) cited Hart who stated that understanding variables involves knowing that the label can store a value that changes. She concluded that programming helps students understand mathematical variables.

Shneiderman (1985) talked about the different uses of variables. He talked about four properties or types of variables: counter variables (for use in loops), counting with a variable, summing with a variable, and general uses of a variable. Across the first three types he raised the issues of

initialization, incrementation, final values, and forming totals. When Shneiderman talked about the general uses of a variable he addressed issues such as assignment statements and the difference between the variable and the value stored in the variable, printing, using, changing, and comparing the value stored in a variable as well as the different types of variables (integer, float, character).

In their study Bayman and Mayer (1983) provided misconceptions about two different types of assignment statements. One statement was an initialization (LET D = 0) and the other was an equation (LET A = B + 1). In both of these statements students often thought the computer was writing the information somewhere or printing it to the screen as opposed to storing it in a specified memory location. Some students believed that the computer stored the equation as opposed to a value. The authors concluded that "Beginning programmers need explicit training concerning memory locations and under what conditions value stored in those locations get replaced" (p. 679).

Perkins and Simmons (1988) talked about a misconception that students may have about the names of variables. They stated:

The notion that a computer program 'knows' what input values should go into variable names like LARGEST dies hard, even though students know in principle that the choice of variable names in theirs, a point incoherent with such wisdom on the part of the computer. Such examples suggest that people commonly fail to notice that incoherencies in their intuitive mental models, and often when incoherencies are brought to their attention, the incoherencies simply do not appear very important. The robust intuitive model seems worth preserving in the face of a few minor discrepancies. (p. 312)

Canas et al. (1994) used a common problem novice programmers have in their study. They assessed students' ability to determine the appropriate data type for a variable and whether the student used the variable correctly.

(Samurcay, 1985) looked at four ways variables are assigned values through assignment statements:

1. assignment of a constant value -- $a = 3$;
2. assignment of a calculated value -- $a = 3 * b$;
3. duplication -- $a = b$;
4. accumulation -- $x = x + 1$;

He continued to describe how each of these may be used within two contexts: external and internal. External variables are inputs to and outputs of the program. These are under control of the program user. Internal variables are necessary only for the solution of the problem and are controlled by the programmer. The author believes that internal variables will be harder for novice programmers. In order to look at this, the uses of a variable in a loop are explored. Three types of variable processes are involved in loops:

1. update - accumulation variable
2. test - condition for terminating the loop
3. initialization - initial values of loop

Findings suggested that initialization is more difficult than updating or testing.

References

- Bayman, P., & Mayer, R. E. (1983). A Diagnosis of Beginning Programmers' Misconceptions of BASIC Programming Statements. *Communications of the ACM*, 26(9), 677-679.
- Canas, J. J., Bajo, M. T., & Gonzalvo, P. (1994). Mental models and computer programming. *Journal of Human-Computer Studies*, 40(5), 795-811.
- Du Boulay, B. (1986). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2(1), 57-73.

- McCoy, L. P. (1990). Literature Relating Critical Skills for Problem Solving in Mathematics and Computer Programming. School Science and Mathematics, 90(1), 48-60.
- McCoy, L. P., & Burton, J. K. (1988). The Relationship of Computer Programming and Mathematics in Secondary Students. 159-166.
- McGill, T., J., & Volet, S. E. (1997). A Conceptual Framework for Analyzing Students' Knowledge of Programming. Journal of Research on Computing in Education, 29(3), 276-297.
- Perkins, D. N., & Simmons, R. (1988). Patterns for Misunderstanding: An Integrative Model for Science, Math, and Programming. Review of Educational Research, 58(3), 303-326.
- Samurcay, R. (1985). The Concept of Variable in Programming: Its Meaning and Use in Problem-Solving by Novice Programmers. Education Studies in Mathematics, 16(2), 143-161.
- Shneiderman, B. (1985). When Children Learn Programming: Antecedents, Concepts and Outcomes. The Computing Teacher, 12(5), 14-17.
- Spohrer, J. C., & Soloway, E. (1986). Alternatives to Construct-Based Programming Misconceptions. Paper presented at the CHI'86 Proceedings.