DOCUMENT RESUME

ED 432 240 IR 019 610

AUTHOR Galloway, Jerry P.
TITLE The Value of Programming in Beginning Educational Computing.
PUB DATE 1999-03-00
NOTE 7p.; In: SITE 99: Society for Information Technology & Teacher Education International Conference (10th, San Antonio, TX, February 28-March 4, 1999); see IR 019 584.
AVAILABLE FROM Web site: http://ww2.netnitco.net/users/jpgtma/papers.htm
PUB TYPE Reports - Evaluative (142) -- Speeches/Meeting Papers (150)
EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS Computer Literacy; *Computer Science Education; Computer Uses in Education; *Course Content; Curriculum Development; Educational History; Elementary Secondary Education; Higher Education; Holistic Approach; Introductory Courses; *Preservice Teacher Education; Problem Solving; *Programming; Skill Development; Thinking Skills; Training

ABSTRACT

This paper discusses in detail the nature of the conceptual development in beginning computing education for teachers and makes a case for the inclusion of programming experiences. The controversial nature of this perspective is addressed directly including a historical perspective. The discussion includes an account of some specific notions of computer operations, problem-solving skills, and sound computing concepts that warrant programming experiences. The nature of holistic learning for preservice teachers, the concepts of education versus training, and the demands on educators for mastery, problem-solving, and adaptability in a quickly changing world of technology are issues brought to bear on the argument in favor of programming experiences in teacher education. (Contains 19 references.) (Author/AEF)

The Value of Programming in Beginning Educational Computing

Jerry P. Galloway
Associate Professor of Education
Coordinator of Computer Education
Indiana University Northwest
3400 Broadway - Gary, Indiana 46408

ABSTRACT: This paper discusses in detail the nature of the conceptual development in beginning computing education for teachers and makes a case for the inclusion of programming experiences. The controversial nature of this perspective is addressed directly including a historical perspective. The discussion includes an account of some specific notions of computer operations, problem-solving skills and sound computing concepts that warrant programming experiences. The nature of holistic learning for preservice teachers, the concepts of education versus training, the demands on educators for mastery, problem-solving and adaptability in a quickly changing world of technology are issues brought to bear on the argument in favor of programming experiences in teacher education.

Background

One of the most controversial issues in the early computer education of pre- and inservice teachers is the inclusion of programming. This debate has raged since the earliest years of preparing teachers (and preservice teachers) to use computer technology. In the late 1970's and early 1980's as more teachers began to use word processing, many teacher educators argued that computer literacy should involve little more than word processing skills. Even at that time, when the availability of user-friendly applications was relatively scarce and programming was commonplace among "techies," its inclusion in beginning educational computing courses was controversial and opposed by many.

In the areas of computer science or data processing and information systems, students learn a number of programming languages as well as programmable database management programs. Nevertheless, such uses of computers, while considered essential and commonplace in those related disciplines, are typically opposed by educators of all sorts. Reasons for such opposition have varied somewhat over time. In the early 1980's opposition to the inclusion of programming experiences included arguments that programming is too difficult. Arguments even claimed that too few educators were really qualified to teach programming which seems like a self-fulfilling prophecy if educators never learn it.

However, most such arguments have been left in the past with the early periods of computer literacy movements. Today's opposition is somewhat different. Goals have changed. As information systems technology, instructional technology and educational computing have evolved into legitimate disciplines and as school systems and state education departments throughout the country have long focused on the integration of technology into education, computer literacy is considered passé and programming is often ignored or discredited.

Both in the earlier years and since, another factor has greatly influenced the argument. Educators simply do not want to do programming. It is considered too difficult, very confusing, too time consuming and educators fail to see any value in the applied, everyday usage of computers. Much like a child resists eating vegetables because they're distasteful and then fashions argument after argument espousing the merits of alternative diet options like peanut butter, educators have rationalized the exclusion and avoidance of programming as a useless endeavor and a distraction to more important areas of educational computing. However, these attitudes against programming, the resistance and inhibition of educators, do not constitute a worthy rationale against the value of programming - especially as a mere experiential component of early computer education.

Early rebuttals in favor of programming were generally weak and ineffective. The argument that programming would remain an inevitable part of computer use was never accepted and today seems even less believable. The argument that important educational software would be left to non-educators to develop or that, at best, an awkward cooperation between educators and computer programmers would be required was always a better argument. But, this too dissolved into the past as educators have continued to be satisfied with and accept the computing world as it has been given to them. That is, educators have been satisfied being mere users of computers and software allowing the real nature of their environment to be determined by computer scientists.

Programming

So, why should programming be included in beginning educational computing courses? And, if so, what sort of programming experiences? The notion of preparing educators who, with good programming concepts, can deal

with the development of new software has considerable merit but may still not convince teacher trainers focused on classroom issues. The idea that teachers do not do programming and thus have no need for the training is weak considering that there are different types of programming today. A number of common software programs involve programming skills (often called scripting) and also involve programming concepts (structure and design) all perfectly suited to the classroom teacher. Nevertheless, this argument is also frequently rejected, as educational computing students still do not want to learn programming.

Nevertheless, the real task is to prepare teachers to function successfully in the highly technological world that is developing. They, in turn, must be prepared to lead their students into computing, that is, to acquire the necessary skills to survive in our society. While many educators believe that programming has no place as beginners focus on the so-called tools of the applied utilities of productivity software, Jones (1990) specifically states that virtually all students in gifted and talented programs should be taught programming. It is suggested that learning programming languages aids students in learning to use computers as real tools. It is further suggested that the real value of computing lies in the problem-solving experiences of posing and answering questions. LOGO is one programming language claimed to provide opportunities to help students learn to integrate knowledge and share ideas.

Others have called for something different than the traditional skills of application software for beginning computing students. Eisenberg and Johnson, (1996), advocate an information processing model which specifically calls for computer programming. The point is to develop students with greater problem-solving and higher-order thinking skills. This, of course, stems from a perspective about what skills and knowledge teachers actually need and what really constitutes computer literacy. A great deal of literature has well documented the relationship between programming and improved problem-solving skills.

While usually a programming language is one avenue to improving problem-solving skills, Milbrandt (1995) documents the reverse: using problem-solving techniques to learn programming. Students learning computing concepts through a gradual increase in problem difficulty. Accounting for the importance of programming structures in computer studies, Milbrandt specifically calls for computing courses to incorporate as many programming language experiences as possible.

However, the exclusive focus on programming for teachers is clearly a thing of the past, and rightly so. Allan and Kolesar (1996) acknowledge the importance of including applications and tool-based skills of traditional courses, but such experiences are still suggested as preparatory for the programming experiences for beginners. It seems obvious that the quickly changing world of technology, the often challenging nature of computing tasks and the problems to which they are directed testifies to the importance of problem-solving skills. The conceptual development, improvement of problem-solving and higher-order thinking skills in computing have been directly linked to the inclusion of Logo programming (Allen, 1993; Battista, 1994; Borer, 1993; Dalton & Goodrum, 1991; Kommers, 1995) and BASIC programming (Overbaugh, 1993).

Many of the old arguments against programming will remain in the past where they belong. It is instead important to address the nature of computing and its demands for the conceptual development of beginning computing students. Ideally, technical details of programming can be identified in terms of benefits to improving understanding. The function of and the nature of characters in word processing are viewed differently with a programming perspective. The importance or relevance of data is presented differently in programming than in word processing. Handling different types of data (numeric / text) is necessary in using a database and spreadsheets and programming experience can directly impact those skills. Packaged functions in programming, the concepts of command, language, data, etc., the expandability of command sets (newly defined procedures in DOS or LOGO compared with Macros in word processing, etc.) - all support a conceptual relationship between experiences in programming and skills/performance using common applications software.

There is a convincing argument that many uses of computers today do in fact involve programming and that, as such, "programming" is being redefined, and thus effective use of many applications does require a well-developed programming perspective. For example, HyperCard is fashioned as a user-based application. That is, it is designed to appear like a database program (fields, records, etc.) for computer users when it is actually a type of programming language. HyperCard, HyperStudio, Toolbook, other authoring systems and Web page design, not to mention the highly technical task of programming HTML code - all very much involve programming concepts, skills and demand a programming mentality.

Some comparisons have shown that BASIC programming can out perform HyperCard programming for improving problem-solving skills. While HyperCard was superior in product development, BASIC is the preferred tool for improving problem-solving skills (Reed & Liu, 1992). The right approach to learning BASIC programming can help improve one's ability to understand and solve problems (Tsai, 1992). Henry and Southerly (1992) detail what is essentially an inappropriate comparison of attempting in BASIC, that which was done in HyperCard. After all, they are two different kinds of languages best suited to the production of different kinds of products. Nevertheless, both were deemed to contribute to the cognitive outcomes of improved problem-solving skills (HyperCard being the more user-friendly of the two).

The inclusion of programming in beginning educational computing courses, while misunderstood and resisted by many, will make educators better computer users, better teachers of computing and better able to assist students

with computing responsibilities. The improvement of critical thinking and problem-solving skills in educational computing is supported by another perspective.

Education vs. Training

The most common and seemingly worthy argument today against programming is that teachers will simply not use it. BASIC programming, while still an evolving, state-of-the-art high-level programming language, is disparaged and discredited as an ancient and useless pass-time - a distraction and pointless activity for modern educational computing agendas. LOGO is often misunderstood and widely ignored. Pascal, while often respected more by the computer science world, has also been dropped from many educational training programs. After all, educators do not have the time to program and, as any survey or field observation would show, most computer-using educators do not do programming.

While this might be true in one sense, it is not the proper issue. Surely a college education should produce more than simply memorized procedures and tasks accomplished. "A meaningful, unified computer literacy curriculum must be more than 'laundry lists' of isolated skills," (Eisenberg & Johnson, 1996, p. 1). It's impossible to only teach what will be used. The general computing skills of using the various parts of a computer system, managing lists of information, writing and creating products using word processors and other applications, do not help students transfer and apply skills in changing situations.

The inadequacy of mere procedural rituals, without a deeper conceptual understanding, has been examined throughout many years of research on educational computing (Galloway, 1992; Hobbs & Perkins, 1985; Kintsch & Greeno, 1985; Mawby, Clement, Pea, & Hawkins, 1984; Mayer, Bayman & Dyck, 1987; and Perkins & Simmons, 1988). In other words, a complete education is more than the sum of its parts whereas training consists only of parts soon to become obsolete. If concepts for beginning computing students (as compared to competencies) are not obsolete then neither are programming experiences which can directly impact that conceptual development.

Teaching the Whole Person

It is a common and dangerous misunderstanding in departments and schools of education that preservice educators acquire a liberal arts education and worldly knowledge from the other experiences in their undergraduate curriculum. It seems that educators far too often believe that such responsibilities fall on the other departments and schools in the university to educate the student leaving the teacher training program to focus on technical skills and mere competencies as if pedagogy is little more than a science of formulas to be mechanically applied. Education at the university level (especially for undergraduates), as distinct from the technical school or junior college, demands a more broad-based holistic program. The mere training of high school graduates to perform specific tasks without a more elaborate and general education will not produce competent and desirable teachers.

Today's naiveté, ignorance and misconceptions of beginning computing students is no less significant and fundamental than 20 years ago. Students have misconceptions about numbers, how to think about information, how to reason. Students lack perspectives which must be generated and cultivated. Today's teacher trainers must not simply train - they must educate. In this endeavor, it is important for college-level computer educators to teach about numbers, to teach about communication and thinking, to teach about organization and planning, to produce or pursue in a more holistic fashion that ephemeral and elusive thing called a complete education. The value of programming to contribute directly and significantly to that noble goal should not but is in fact regularly ignored, overlooked and undermined. The essential point about the inclusion of programming for beginning educators is not to produce programmers per se but to allow programming experiences to better educate computer-using teachers.

The remainder of this paper will focus on identifying some aspects of a programmer's perspective in contrast with that of a user that could impact the conceptual development of beginning computing students. Consider the following notions.

The Programmer's Perspective

Computer users tend to take the features of their computing environment for granted. For example, if a software product does not greet the user by name or display today's date conveniently at the top of the screen, users tend to accept such characteristics without question. A programmer on the other hand will more likely question the nature of the environment and seek to customize or modify available features. The distinction may often be a subtle matter of perspective but seeing and manipulating the basic commands of a programming language can help break down the mystique of computer operation. The anthropomorphization of using a computer is the misconception of the user to which programmers are immune.

There is a kind of designer mentality as programmers write, develop and debug programs. They plan for the deferred use of their products by the users for whom they labor. While the users see only the puppets on the stage performing the show, the programmers see the strings, the puppeteer and back stage construction of the set. Indeed, they design the show. This (programming experience) can make, not a subtle, but a profound difference in the perspective and mentality of a computer user.

There are a number of technical experiences in programming that can enhance computer use. The relative safety of "user"-mode fails to demand the proper concern for precision, planning and organization, simple good habits which can improve computer use. Errors can often be undone. It is not necessary remember commands as menus typically provide the opportunity to browse and guess until the solution is inadvertently discovered. However, while user-mode is relatively safe in the more modern software programs, there are wide ranges of use from effective and efficient to the awkward and inept.

What is a comma? What is a semi-colon? Etc. To many users, they are simply the symbols of sentence punctuation. To programmers and computers alike, such symbols are often the technical components of data manipulation, parameter specification and command structure. Too, their roles and function can vary from situation to situation. Programmers expect this. In programming, commands require specific syntax and structures. The exact nature of instructing the computer is in sharp contrast to the automatic help and auto-correction of errors in using modern commercial software.

Users typically create, manipulate and focus on the development of data. The notion that such data is virtually impotent and of little consequence to the computer is an interesting perspective. However, in programming this is commonplace. For example, in a program one may display either "Kat" or "Cat" and obviously the computer doesn't care. Users can often misunderstand that computers do not truly "know" about how words are spelled or what is intended in content - in spite of identifying misspellings with spelling checkers. In fact, the emulation of artificial intelligence by today's computers, so-called "smart" machines, can only exaggerate the anthropomorphic misconceptions of beginners mentioned earlier.

Exploring different "types" of data allows one to consider both the incidental and arbitrary nature of characters (as far as the computer is concerned) compared with the relatively potent and functional nature of numeric values. That is, "47" (mere characters) can be used for nothing but display and storage whereas 47 (the numeric value) can determine the number of times things occur or where or how something might happen. These notions, along with how numerals can be used as proper names (X19, R2D2) rather than always indicating numeric quantity, is uniquely examined and practiced in programming and is easily overlooked in the world of the user.

Packaged functions (UCASE$) can be considered an expansion on a language (set of available commands) and thereby constitute a variation on the simpler notion of command. Programming allows one to infinitely expand the limited set of commands provided in a language. For example, declaring subroutines in BASIC or Pascal create addition components to the basic language. Writing procedures in Logo creates additional commands, and while they're distinguished from primitives (the original set of embedded commands), they nevertheless constitute a personalized expansion on the function and capabilities of the computer as these new "commands" are added. Writing batch files in DOS similarly expands the base set of commands in the language and thus customizes the environment and develops computing power. Experience in programming allows one to relate to this computing power in important ways. The user-mode equivalent might be creating macros in word processors or spreadsheets, designing queries in a database or using any authoring tool (hypermedia or web design).

Beginning students, in general, seem to have difficulty with the concept of representation. That is, simply, how it is that a label can be made to represent something else and then be used in its place. Students often get confused using the label in stead of the thing being represented. While this is certainly important in using variables in algebra and computing alike, it is also a problem for students in common language. Studying the use of variables in computing (or algebra) can be a vehicle for assisting students with representation in language and communication.

For example, students often confuse a term like COMPLETE with a term like "COMPLETE" (when in quotes). That is, the term when used normally has a specific definition, in this case, meaning something in its entirety, whole, total, or finalized and finished. However, once placed in quotes, the term no longer means the same thing. That is, the term in quotes is used as a kind of label to represent something which is in actuality not whole or finalized at all, but which will be accepted as such. For example, I was reading a book and I stopped once my reading was "complete." This would of course mean that the reading actually terminated before the end of the book. It might mean that all interesting ideas in the book were covered and, that being sufficient, the reading was finished and considered "complete." Of course, it does not mean that the reading in any way truly covered all words or all pages. Students will frequently fail to understand this distinction and how terms can be used as labels to represent other ideas.

Variables in programming allow students to work with data through their representative labels. An astute instructor can present variables in ways to force students to focus on more than their function in an algorithm and teach students about the notion of representation itself. For example, a numeric variable in BASIC (B3) could have as its value the quantity of five (5). While another numeric variable (B5) could have as its value a quantity of three (3) - (B5=3, B3=5). Also, a string variable YES$ could equal "NO" while another string variable NO$ could equal "YES". Using these variables, these labels which represent other data, can help lead students to explore important

aspects of using language. Consider responding with the appropriate string variable to answer "Does B5 plus 2 equal 7 ?"

The arbitrary nature of labels, (it has been said "what's in a name?"), is important in computing, logical thinking and problem-solving in general. To what extent does the range of representation extend? It is a temporary or local representation or more global affecting other procedures and routines. This is certainly relevant in HyperCard scripting (a task often taken on by non-programmer educators). The view that spreadsheet cells are actually variables taking on values as data are entered is a perspective lost to non-programmers.

Design issues frequently arise in a number of hypermedia authoring packages like HyperCard, HyperStudio, Toolbook or other programs that are supposedly designed for the average "user" and provide an automated system of controls for creating packaged experiences. Users are typically made to believe that this is not programming whereas programmers not only understand how this is programming but may even seek a greater and more direct control of the language or authoring tool involved.

Loops in programming offer opportunities for students to explore how situations can be manipulated with numeric values. Recently this author asked students to name all even numbers between 1 and 10 (inclusive). Students correctly responded: 2, 4, 6, 8, 10. Then students were asked to count from 1 to 10 by two's. Students erroneously replied again with 2, 4, 6, 8, 10. (Correct answer is of course: 1, 3, 5, 7, 9.) The logic of program flow, the way numbers affect situations are all important aspects to using computers which can be experienced in programming.

IF/THEN logic is often lost on many people - not the least of which are beginning computing students. For example, "if mom comes home then I'm going to the store." Such a hypothetical is easy for most when the antecedent (mom comes home) is true. However, when students are asked what happens in the case of mom not coming home, they often answer "I don't know but you're not going to the store." When they are told that "I'm going to the store anyway" they often fail to understand the logic. After all, the original hypothetical did not say "if and only if mom comes home...." In fact, it never addressed what happens if mom does not come home one way or the other. Such basic elements of thinking and problem-solving are easily explored and experienced in programming. The complete flow of programs which branch from section to section based on this or that force programmers to learn logical consequences and improve reasoning skills. Computer users can benefit from such education (as distinct from training) and would be better users because of it.

There are many aspects to what programmers experiences which have a direct impact on the use of applications software. For example, users view everything but displayable characters in a word processor as "space." The real nature of a space (ascii code 32) as a real character is a fundamental aspect of data in programming. Users view commas and semicolons as sentence punctuation symbols whereas they are data delineators for programmers. Formatting fields in a database or cells in a spreadsheet is a natural part of handling data types in programming. Analyzing the content of a display would involve different considerations from a programmer. - a different perspective - than found in a computer "user." This perspective is often a critical difference between users who do and do not have programming experience.

In summary, arguments against programming actually fail to support the total avoidance of programming just as pro-arguments do not suggest that beginning courses should consist entirely of programming. The issue is not to distinguish programmers from users but to distinguish users with programming experience from those with none. Beginning computer education should include at least a brief programming experience to help build stronger concepts, adaptability, and problem-solving skills for our future educators.

This paper and other papers can be found at the author's personal web site for papers....
http://ww2.netnitco.net/users/jpgtma/papers.htm

References

Allen, J. (1993). The impact of cognitive styles on the problem-solving strategies used by preschool minority children in Logo microworlds. *Journal of Computing in Childhood Education, 4* (3-4), 205-217.

Allan, V. H.; Kolesar, M. V., (1996). *Teaching computer science: A problem-solving approach that works.* Proceedings of the Annual National Educational Computing Conference, Minneapolis, MN, June 1996.

Battista, M. T. (1994). Research into practice: Calculators and computers: Tools for mathematical exploration and empowerment. *Arithmetic Teacher, 41* (7), 412-417.

Borer, M., (1993). *Integrating mandated Logo computer instruction into the second grade curriculum.* (ERIC Document Reproduction Service No ED367311).

Dalton, D. W., Goodrum, D. A., (1991). The effects of computer programming on problem-solving skills and attitudes. *Journal of Educational Computing Research, 7* (4), 483-506.

Eisenberg, M. B.; Johnson, D., (1996). *Computer skills for information problem-solving: Learning and teaching technology in Context.* (ERIC Document Reproduction Service No ED392463).

Galloway, J. P. (1992). *Analogies in educational computing.* East Rockaway, NY: Cummings & Hathaway.

Henry, M. J., Southerly, T.W., (1992). *A comparison of the language features of BASIC and HyperCard.* Paper presented at the Annual Conference of the Easter Educational Research Association, Hilton Head, SC. March, 1992.

Hobbs, R., & Perkins, D. (1985). Rituals and models: Students' understanding of the computer. Concept Paper: Cambridge, MA: Educational Technology Center, Harvard Graduate School of Education.

Jones, G., (1990). *Personal computers help gifted students work smart.* ERIC Digest #E483. (ERIC Document Reproduction Service No ED321488).

Kintsch, W., & Greeno, J. G. (1985). Understanding and solving word arithmetic problems. *Psychological Review, 92,* 109-129.

Kommers, P. A. M. (1995). *Conceptual Design to complement hypermedia as learning tools.* (ERIC Document Reproduction Service No ED385238).

Mawby, R., Clement, C. A., Pea, R. D., & Hawkins, J. (1984). *Structured interviews on children's conceptions of computers. (Technical Report No. 19).* New York, NY: Bank Street College of Education.

Mayer, R. E., Bayman, P. & Dyck, J. L. (1987). Learning programming languages: Research and applications. In *D. E. Berger, K. Pezdek, & W. P. Banks, (Eds.), Applications of cognitive psychology: Problem-solving, education, and computing.* Hillsdale, NJ: Erlbaum.

Milbrandt, G. (1995). Using problem-solving to teach a programming language. *Learning and Leading with Technology, 23* (2), 27-31.

Overbaugh, R. C., (1993). *A BASIC programming curriculum for enhancing problem-solving ability.* (ERIC Document Reproduction Service No ED355921).

Perkins, D. N., & Simmons, R. (1988). An integrative model of misconceptions. *Review of Educational Research, 58* (3), 303-326.

Reed, W. M., Liu, M., (1992). *The comparative effects of BASIC programming versus HyperCard programming on problem-solving, computer anxiety, and performance.* Paper presented at the Annual Conference of the Eastern Educational Research Association, Hilton Head, SC. March, 1992.

Tsai, S. (1992). *Development of schema knowledge in the classroom: Effects upon problem representation and problem solution of programming.* Paper presented at the Annual Conference of the American Educational Association, San Francisco, CA, April 20-24, 1992.

U.S. Department of Education
Office of Educational Research and Improvement (OERI)
National Library of Education (NLE)
Educational Resources Information Center (ERIC)

NOTICE

REPRODUCTION BASIS

☒ This document is covered by a signed "Reproduction Release (Blanket) form (on file within the ERIC system), encompassing all or classes of documents from its source organization and, therefore, does not require a "Specific Document" Release form.

☐ This document is Federally-funded, or carries its own permission to reproduce, or is otherwise in the public domain and, therefore, may be reproduced by ERIC without a signed Reproduction Release form (either "Specific Document" or "Blanket").