Paradigms and laboratories in the core computer science curriculum: An overview

Hartel, P.H.; Hertzberger, L.O.

Publication date
1995

Published in
SIGCSE bulletin

Paradigms and laboratories in the core computer science curriculum: An overview

Pieter H. Hartel * L. O. Hertzberger *

Abstract

Recent issues of the bulletin of the ACM SIGCSE have been scrutinised to find evidence that the use of laboratory sessions and different programming paradigms improve learning difficult concepts and techniques, such as recursion and problem solving.

Many authors in the surveyed literature believe that laboratories are effective because they offer a mode of learning that complements classroom teaching. Several authors believe that different paradigms are effective when used to support teaching mathematics (logic and discrete mathematics) and computer science (programming, comparative programming languages and abstract machines).

Precious little evidence by way of reported results of surveys, interviews and exams was found in the ACM SIGCSE bulletins to support these beliefs.

1 Introduction

Over the years, computer science (CS) has evolved from a single discipline into a number of related disciplines. The 1991 ACM/IEEE-CS Computing Curricula report [29] identifies a number of subjects that can be classified either as belonging to the core of CS or being peripheral.

Core CS consists of programming languages, algorithms and data structures, architecture, operating systems, and software engineering. The Computing Curricula recommendation devotes just under 90% of the lecture hours to these core subjects.

*Department of Computer Systems, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands pieter@fwi.uva.nl

A host of non-core (peripheral) topics are now firmly established. These include databases, artificial intelligence, symbolic and numeric computation and human-computer interaction. The Computing Curricula report recommends that just over 10% of the lecture hours should be spent on these peripheral subjects.

The number of applications of CS is on the increase. This has caused two separate trends. Firstly, some CS curricula are being expanded to contain more peripheral CS topics. Secondly, many non-CS curricula now include the study of some CS topics. Many departments that started out as pure CS departments have acknowledged these developments and are now offering a number of different curricula, often in combination with subjects such as psychology, business studies, law, medicine, languages, physics etc. While previously a university would offer only one CS curriculum, presently many universities offer several curricula with a CS component.

The diverging trends in CS curricula necessarily have an impact on the teaching practice of the CS departments. This divergence brings with it a demand to focus the various curricula. Aspects of this focusing activity will be discussed here. To simplify the discussion, hopefully without being unjust, we claim that core CS is intimately related to discrete mathematics and logic. Where a relation exists, the peripheral CS disciplines are more closely associated with other branches of mathematics such as calculus, or probability theory.

The relationship between logic and discrete mathematics within the core CS curriculum will be highlighted as found in the papers being surveyed. All issues of ACM SIGCSE bulletin over the past seven years have been scrutinised to find evidence of this relationship.

An important element of the above relationship is the emphasis placed on learning through experimentation. The same literature has been scanned to find descriptions of how laboratory sessions are used to support learning. Some 30 papers were found to contain relevant information.

The present survey is restricted to papers published in SIGCSE, hoping to provide a view reflecting that generally held by those with an interest in education proven by publication in this medium.

The next section reviews the papers that discuss general issues. Section 3 covers papers that use programming in various paradigms and/or laboratories for teaching discrete mathematics and logic. Teaching core CS topics using laboratories and various paradigms is discussed in Section 4. The last section presents some discussion and the conclusions.

2 General issues

The general issues of relevance that were found in the literature are abstraction, recursion and learning with the aid of laboratories.

2.1 Abstraction and generalisation

Návrat [17] argues that teaching abstraction and generalisation as problem solving concepts can be supported by the use of appropriate programming paradigms. His point is that three methodological dimensions must be distinguished: abstraction and concretisation, generalisation and specialisation, and meta-knowledge. The first two dimensions are often confused. For example, an abstract solution to a problem should be no more general than a concrete solution. The third dimension, that of meta-knowledge, captures programming tools and techniques that can be applied in specific circumstances. At various points in the three dimensional space thus established, Návrat finds that specific programming paradigms are particularly useful for highlighting the distinctions between the three dimensions.

Along the same lines, Jarc [10] presents a unified view of data structures, which can be recognised as an object hierarchy. In both papers [17, 10] programming in different paradigms is mentioned as a means to make concepts operational.

2.2 Recursion

It is essential to teach recursion as a concept in its own right. It is often perceived as a difficult concept so it must be taught properly. Firstly, learning recursion requires a new mode of thinking. Secondly, many important algorithms are recursive.

Wiedenbeck [32] reports that to provide extensive examples for students to emulate is vitally important. Few such examples are available in the real world. The literature offers two approaches to learning recursion: plan-based recursion and recursion as a problem solving tool.

2.2.1 Plan-based recursion

According to Newmarch [18] a plan-based approach to recursion presupposes a solution to a given problem. The solution is then matched to a repertoire of plans that describe the basis of the recursion and the recursive step(s). Examples are 'linear recursion' and 'divide-and-conquer'. Newmarch makes plans operational using Prolog. A small number of plans support a large number of different programs and algorithms. The plan-based approach also works with the imperative programming paradigm.

2.2.2 Recursion as problem solving tool

Instead of presupposing a solution, recursion can be presented as a problem solving tool. Henderson [8] claims that it is natural to do so in the declarative paradigm (he uses SML). In that paradigm, recursion can be taught to go hand-in-hand with induction in discrete mathematics. The importance of laboratories using SML to reinforce the use of recursion as a problem solving concept is stressed.

Many imperative programming languages require the use of pointers to manipulate recursive data structures effectively. Pointers are a relatively low-level concept. They are therefore mostly taught as an advanced concept, late in a programming course, when problem solving has already been taught. The use of an imperative programming language therefore makes it difficult to teach recursion and recursion-based problem solving early.

Fractals provide a good example for teaching recursion [5]. Recursion is shown to be present at three levels: at the visual level (fractals are self-similar structures), at the analytic level (dimensionality analysis plays an important role in work with fractals) and in the structure of the programs that draw fractals (for which Prolog is used). The recursion at each level is essentially the same, thus reinforcing the concept. The visual aspect stimulates exploratory learning of recursion.

2.3 Laboratories

Laboratories support the learning process by offering students well-chosen, short, well-paced exercises. Laboratory assignments:

- make the material being studied operational
- allow the student to ascertain that the material is understood
- provide instant feedback to the student
- stimulate experimentation with the material
- raise questions for further lectures and study
- offer the student the opportunity to discover solutions to problems

Various authors [22, 14] argue for an experimental approach to the CS curriculum, using programming assignments. One author goes further: Biernat [2] in his course on algorithms and data structures uses puzzles and games for class room teaching and home work. He observes that students understand algorithms faster, remember them better and are excited about the course.

3 Discrete mathematics and Logic

Henderson [7] argues that a course on discrete mathematics should precede the first programming course. He claims that mastering mathematical concepts, such as recursion and induction, is a requirement for learning proper programming. The discrete mathematics course described by Henderson concentrates on general problem solving principles, such as patterns and symmetries, and recursive and inductive thinking. Course material is reinforced through exploratory computer-based laboratory assignments. Henderson uses SML, SETL and Trilogy (a logic language). The emphasis in the laboratory is on experimentation with concepts through the use of interpreted implementations of programming languages. This is quite different from learning to program. The distinction between experimentation on the one hand and programming on the other is an important one. The exploratory character of the laboratory stimulates the students and makes mathematics a fun subject.

Wainwright [31] describes laboratory assignments (using Miranda[1]) to support a discrete mathematics course for second year students. The course is taught to CS students, mathematics students and students in Computer Information Systems (CIS). Some students have no programming background, so that laboratory assignments have to be based on languages that are easy to learn. Responses to a survey were received from 13 CS, 9 CIS and 10 Mathematics students. The survey reports that even without prior exposure to programming, Unix or editing with Vi, all students found the laboratory assignments useful and recommend that they be kept as part of the course. The CS students were slightly more enthusiastic than the Mathematics and CIS students.

Schoenefeld and Wainwright [25] describe a summer course in discrete mathematics for teachers at high-school level. This course is taught using the book by Skiena [26], which offers extensive laboratory assignments based on Mathematica. A number of surveys have been held amongst the course participants, indicating that they were "extremely pleased that Mathematica was integrated into the course". The use of Mathematica as an exploratory tool makes operational important mathematical concepts such as abstraction and generalisation.

Myers [15] makes a plea for teaching more logic as part of the CS curriculum. One of the many reasons he gives is that the basis of all important programming paradigms is found in logic.

Hein [6] describes tools to teach discrete mathematics and logic using a large variety of short and relevant laboratory assignments. Simple assignments are described to experiment with laws of logic (using Prolog). Other assignments support learning about functions and function composition (using FP) and learning about types (using SML). The use of different languages is not seen as a problem, as the languages that are used faithfully reflect the notations used in the course and emulate in a natural way the notions of discrete mathematics and logic. Hein notes that a course supported by extra laboratory assignments does require students

[1]Miranda is a trademark of Research software Ltd.

to do more work. One should thus not forget to award more credits for such a course than for a course without a laboratory.

4 Core Computer Science

In the laboratories that support teaching general and mathematical concepts an experimental approach to programming and programming paradigms is used. Teaching core CS subjects requires a different, more fundamental approach to programming and programming paradigms. However, there is again a relationship between particular subjects and the choice of paradigm that is most suited to teach that subject. In the SIGCSE bulletins, the following subjects were discussed with a relationship to paradigms and/or laboratories: the first programming course, the comparative study of programming languages and the study of abstract machines.

4.1 First programming course

Sánchez-Calle and Velázquez-Iturbide [23] teach the declarative paradigm (using Hope+) in the first programming course, with a strong emphasis on synthesis and analysis of programs using logic to reason about programs. The authors report, without presenting the evidence, that their students "enjoy building programs while they are also able to analyse them rigorously".

Louden [12] reports on an experiment with two different streams of students. The first stream consists of first year CS students. The other stream consists of more mature students. The CS students learn Logo first, "emphasizing the role of functional programming in a manner similar to Abelson and Sussman [1]". They then learn Pascal. The mature students frequently bypass the Logo and Pascal-based classes. Both streams then learn Lisp. Does the declarative (Logo) background improve the examination results for the Lisp course? Possibly: based on the responses received for a survey (69 CS and 100 mature students), it was found that the examination results of the mature students were slightly better. Louden points out that the relative maturity of this stream might explain this effect and concludes that more study should be done.

Clancy and Linn [3] report on their experience with students who were taught Lisp as a first language. During the subsequent data structures course, these students produce "much better code than their procedurally-brainwashed class mates". The paper does not provide supporting evidence for this rather strong and tendentious claim.

Stanchev and Radensky [23] teach functional programming using an FP-like language to graduate and under-graduate students. They report that their course was found difficult. This is a questionable conclusion for it is based on the number of students that actually took the exam (15 out of 26 the first year, 7 out of 14 the next). The authors hold responsible the "one word at a time" thinking habits induced by prolonged exposure to imperative languages.

The evidence found thus far in the SIGCSE bulletins, for or against using different paradigms in the first programming course barely transcends the anecdotal. A more comprehensive survey would have to include a vast amount of other literature, such as special issues of various journals [28] that are devoted to education.

4.2 The comparative study of programming languages

The comparative study of programming languages is moving away from particular languages towards the study of paradigms. in much the same way as the core CS field as a whole is moving in that direction. In his survey, King [11] notes that one of the goals of a comparative study must be to prepare the students for the future. This implies that some historical perspective and a deep understanding of current concepts are essential. The comparative study of programming languages thus becomes more fundamental [13], as a deep understanding of paradigms is only possible with an understanding of the foundations of the paradigms.

One way to teach comparative programming languages is via their formal semantics. Nielson and Nielson [19] provide a thorough introduction to the subject, supported by many exercises, both using pencil and paper and using Miranda for laboratory assignments. This subject is worthy of study as a separate, but perhaps advanced subject, but it also has its place in the comparative study of programming languages. As part of a course on programming languages, Höft [9] presents the semantics of the loop-construct in a precise way using Dijkstra's calculus of guarded commands. The material is supported by declarative laboratory assignments using Mathematica. The graphical capabilities of Mathematica in particular are used to reinforce learning by studying possible behaviours of non-deterministic semantics. The increased precision offered by Höfts approach gives deeper insights than those offered by the usual natural language descriptions.

4.3 Abstract machines and translations

Abstract machines and translations between abstract machines are topics that appear in such diverse contexts as computer architecture, semantics of programming languages, compiler construction and the theory of computation. A thorough treatment supported with well-constructed laboratory assignments will therefore be beneficial to all of these subjects at once. Such recurring concepts are identified as extremely useful in the ACM curriculum [29].

Some authors describe abstract machines in terms of declarative programs. Prolog [4] even enables 'reverse execution' (i.e. given a finite state machine and its output, the Prolog system will reconstruct the possible inputs). Piotrowski states that Miranda [20] is less versatile, but offers more elegant and robust (typed) specifications of abstract machines. Both of these declarative approaches to experimenting with abstract machines offer readily executable specifications of formal systems.

The translations between different levels of abstract machines can be presented as straight forward function compositions [21]. The functions being composed retain their validity regardless of the context in which they are used.

Čigas [30] uses abstract machines (finite state machines – FSM) as a schema for developing algorithms. His main point is that the visual representation of an FSM is powerful aid in understanding and analysing problems, with programming solutions (in Pascal) immediately available.

5 Discussion and conclusions

A survey of recent issues of the ACM SIGCSE has been presented, with an emphasis on the use of different programming paradigms and laboratory sessions in the CS curriculum. Discrete mathematics and logic are mentioned by many authors as having an important relation with the core CS curriculum. A summary of the literature may be found in Table 1.

Five years ago Luker [13] wrote that teaching CS means to look to the future. He argued for teaching principles and theory, and against training students to be C programmers. He notes that there exist two schools. The 'theoretical school' would be interested only in teaching principles and theory. The 'practical school' would be interested in training C programmers. Based on what has been found during this literature survey one gets the impression that the situation is improving. On the part of the 'theoretical school' there is a trend towards learning concepts using experimental methods. This stresses the practical aspects of concepts learned and makes them operational in more ways than before. The laboratories are used to reinforce learning and to offer different ways of learning.

On the part of the 'practical school', there is still a certain hesitation to embrace theory but not to the same extent as before. Many would now consider the study of comparative programming languages less important than the comparative study of programming paradigms. The use of different programming paradigms is widely believed to be useful both for their intrinsic interest and for their use in making important concepts operational. Interpreters are often used to support laboratory experiments with concepts from logic and discrete mathematics.

Most authors state that courses supported by short and relevant laboratory assignments are more effective than courses without such laboratories. They argue that the increase is more than that which may be expected simply from requiring more time to be spent on the course. The fact that different people learn in different ways enables some students to learn most from the laboratories and some to concentrate on the pencil and paper aspects of a course.

The search of the SIGCSE bulletins for the support of claims that laboratories are useful and that using different paradigms is beneficial, has yielded precious little evidence. More study is needed in this area.

Experience will tell whether laboratory-supported learning of theoretical concepts will make these concepts sufficiently operational to a large enough proportion of our students to prepare them properly for the future.

Ref.	Year	Where	Subject(s)	Paradigms(s)	Lab.	Stats.
[4]	88	Belfast	Abstract machines	L	yes	no
[5]	88	Michigan	Recursion	L	no	no
[32]	88	Nebraska	Recursion	F,I	no	yes
[8]	89	Stony Brook	Recursion	F	yes	no
[12]	89	San Jose State	First course	F	no	yes
[13]	89	California State	Methodology	C,F,I,O	no	no
[14]	89	City New York	Algorithms	I	yes	no
[20]	89	New South Wales	Abstract machines	F	yes	no
[3]	90	Berkeley	First course	F	no	no
[7]	90	Stony Brook	First course	F,L,S	yes	no
[15]	90	San Antonio	Mathematical logic		no	no
[9]	91	E. Michigan	Programming languages	F,Mathematica	yes	no
[16]	91	Poona	Software engineering	I	no	no
[23]	91	Politècnica Madrid	First course	F	yes	no
[27]	91	Sofia	Programming languages	F	yes	yes
[30]	92	Rockhurst, Kansas	Programming techniques	State machines	no	no
[11]	92	Georgia State	Programming languages	C,D,F,I,L,O	no	no
[21]	92	New South Wales	Abstract machines	F	yes	no
[22]	92	San Antonio	Curriculum	F,I	yes	no
[31]	92	Tulsa	First course	F	yes	yes
[2]	93	AT&T, Illinois	Data struct., Algorithms	Games, Puzzles	no	no
[6]	93	Portland state	First course	F,L	yes	no
[18]	93	Canberra	Recursion	L	no	no
[24]	93	Tulsa	Discrete Mathematics	Mathematica	yes	yes
[10]	94	Maryland	Methodology	F,I	no	no
[17]	94	Bratislava	Methodology	F,I,L,O	no	no

Paradigms:						
	C	=	Concurrent	L	=	Logic
	D	=	Data base oriented	O	=	Object Oriented
	F	=	Functional	S	=	Set-based
	I	=	Imperative			

Table 1: A survey of the papers reviewed.

6 Acknowledgements

We thank Marcel Beemster, Hugh McEvoy, Hans van der Meer and Jon Mountjoy for their comments on draft versions of the paper.

References

[1] H. Abelson and G. J. Sussman. *Structure and interpretation of computer programs.* MIT Press, Cambridge, Massachusetts, 1985.

[2] M. J. Biernat. Teaching tools for data structures and algorithms. *ACM SIGCSE bulletin,* 25(4):9–12, Dec 1993.

[3] M. J. Clancy and M. C. Linn. Functional fun. In D. T. Joyce, editor, *21st Computer science education,* pages 63–67, Washington, DC, Feb 1990. ACM SIGCSE bulletin, 22(1).

[4] D. Crookes. Using Prolog to present abstract machines. *ACM SIGCSE bulletin,* 20(3):8–12, Sep 1988.

[5] B. C. Elenbogen and M. R. O'Kennon. Teaching recursion using fractals in Prolog. In H. L. Dersham, editor, *19th Computer science education,* pages 263–266, Atlanta, Georgia, Feb 1988. ACM SIGCSE bulletin, 20(1).

[6] J. L. Hein. A declarative laboratory approach for discrete structures, logic and computability. *ACM SIGCSE bulletin,* 25(3):19–24, Sep 1993.

[7] P. B. Henderson. Discrete mathematics as a precursor to programming. In D. T. Joyce, editor, *21st Computer science education*, pages 17–21, Washington, DC, Feb 1990. ACM SIGCSE bulletin, 22(1).

[8] P. B. Henderson and F. J. Romero. Teaching recursion as a problem-solving tool using standard ML. In R. A. Barrett and M. J. Mansfield, editors, *20th Computer science education*, pages 27–31, Louisville, Kentucky, Feb 1989. ACM SIGCSE bulletin, 21(1).

[9] H. Höft. Implementation of a non-deterministic loop. *ACM SIGCSE bulletin*, 23(2):24–28, Jun 1991.

[10] D. J. Jarc. Data structures: a unified view. *ACM SIGCSE bulletin*, 26(2):2–8, Jun 1994.

[11] K. N. King. The evolution of the programming language course. In M. J. Mansfield, C. M. White, and J. Hartman, editors, *23rd Computer science education*, pages 213–219, Kansas, Missouri, Mar 1992. ACM SIGCSE bulletin, 24(1).

[12] K. Louden. LOGO as a prelude to Lisp: Some surprising results. *ACM SIGCSE bulletin*, 21(3):35–38, Sep 1989.

[13] P. A. Luker. Never mind the language, what about the paradigm? In R. A. Barrett and M. J. Mansfield, editors, *20th Computer science education*, pages 252–256, Louisville, Kentucky, Feb 1989. ACM SIGCSE bulletin, 21(1).

[14] D. D. McCracken. Three "lab assignments" for an algorithmics course. *ACM SIGCSE bulletin*, 21(2):61–64, Jun 1989.

[15] J. P. Meyers Jr. The central role of mathematical logic in computer science. In D. T. Joyce, editor, *21st Computer science education*, pages 22–26, Washington, DC, Feb 1990. ACM SIGCSE bulletin, 22(1).

[16] R. P. Mody. C in education and software engineering. *ACM SIGCSE bulletin*, 23(3):45–56, Sep 1991.

[17] P. Návrat. Hierarchies of programming concepts: abstraction, generality, and beyond. *ACM SIGCSE bulletin*, 26(3):17–28, Sep 1994.

[18] J. Newmarch. A plan-based approach to Prolog recursion. *ACM SIGCSE bulletin*, 25(2):12–18, Jun 1993.

[19] H. R. Nielson and F. Nielson. *Semantics with applications: A formal introduction*. John Wiley & Sons, Chichester, England, 1991.

[20] J. A. Piotrowski. Abstract machines in Miranda. *ACM SIGCSE bulletin*, 21(3):44–47, Sep 1989.

[21] J. A. Piotrowski. Translation – an introductory exercise. *ACM SIGCSE bulletin*, 24(2):20–28, Jun 1992.

[22] R. E. Prather. Computer science in an undergraduate liberal arts and sciences setting. In *23rd Computer science education*, pages 59–64. ACM SIGCSE bulletin, 24(2), Jun 1992.

[23] A. Sánchez-Calle and J. A. Velázquez-Iturbide. Fun, rigour and pragmatism in functional programming. *ACM SIGCSE bulletin*, 23(3):11–16, Sep 1991.

[24] D. A. Schoenefeld and R. L. Wainwright. Integration of discrete mathematics topics into the secondary mathematics curriculum using mathematica – a summer institute for high school teachers. In B. J. Klein, C. Laxter, and F. H. Young, editors, *24th Computer science education*, pages 78–82, Indianapolis, Indiana, Mar 1993. ACM SIGCSE bulletin, 25(1).

[25] W. Schreiner. Parallel functional programming — an annotated bibliography. Technical Report 93-24, RISC-Linz, Johannes Kepler Univ, Linz, Austria, May 1993.

[26] S. Skiena. *Implementing discrete mathematics – Combinatorics and Graph theory with Mathematica*. Addison Wesley, Reading, Massachusetts, 1990.

[27] S. Stanchev and A. Radensky. Teaching some modern functional programming concepts: an approach based on an extended FP-like language. *ACM SIGCSE bulletin*, 23(4):31–40, Dec 1991.

[28] S. Thompson and P. L. Wadler. Functional programming in education. *J. functional programming*, 3(1):1–115, Jan 1993.

[29] A. J. Turner. A summary of the ACM/IEEE-CS joint curriculum task force report: Computing curricula 1991. *CACM*, 34(6):69–84, Jun 1991.

[30] J. F. Čigas. The art of state. In M. J. Mansfield, C. M. White, and J. Hartman, editors, *23rd Computer science education*, pages 153–156, Kansas, Missouri, Mar 1992. ACM SIGCSE bulletin, 24(1).

[31] R. L. Wainwright. Introducing functional programming in discrete mathematics. In M. J. Mansfield, C. M. White, and J. Hartman, editors, *23rd Computer science education*, pages 147–152, Kansas, Missouri, Mar 1992. ACM SIGCSE bulletin, 24(1).

[32] S. Wiedenbeck. Learning recursion as a concept and as a programming technique. In H. L. Dersham, editor, *19th Computer science education*, pages 275–278, Atlanta, Georgia, Feb 1988. ACM SIGCSE bulletin, 20(1).

***********************Second Language Continued From Page 12*************************

Hymes, D.(1972) "On Communicative Competence" in "Sociolinguistics, Selected Readings", Pride & Holmes (eds.)

Krashen, S et al (1977) "Age, rate and eventual attainment in second language acquisition" TESOL Quarterly, 13, 4, 573-582.

Ledgard, H., Whiteside, J. A., Singer, A. & Seymour, W. (1980) "The natural language of interactive systems" Communications of the A.C.M., 23, 10, 556-563.

Lee, M. P., Peacock, D. & Jeffreys, S. (1989) "dBASE as a first programming language" Collegiate Microcomputer, VII, 2, 111-116.

Lee, M. P., Harrison, A. & Kent, A. E. (1991) "Group projects for the software engineering of knowledge based systems" pp 95-107 IN King, G. A. (ed.) "Software engineering in higher education" Southampton Institute, 1-874011-00-1.

Lee, M. P., Pryce, J.D. & Harrison, A. (1994) "Prolog as a first programming language" pp 275-281 IN King, G. A. et alia (eds.) "Software engineering in higher education" Computational Mechnics Publications, 1-85312-289-0.

Peacock, D., Ralhan, V. K., Jeffreys, S. & Lee, M. P. (1988) "The use of a structured project to teach program development" ACM SIGCSE Bulletin, 19, 4, 10-18.

Peacock, D., Ralhan, V. K. & Lee, M. P. (1988) "A first year course in software design and use" ACM SIGCSE Bulletin, 20, 4, 2-8.

Schank, P. K., Linn, M. C. & Clancy, M. J. (1993) "Supporting Pascal programming with an on-line template library and case studies" Int. J. Man-Machine Studies, 38, 1031-1048.

Shaw, Guise & Reddy (1989) "What a software engineer needs to know: program vocabulary" Software Engineering Institute Technical Report 30, Carnegie-Mellon University, Pittsburgh.

Stern, H. H., (1986) "Fundamental Concepts of Language Teaching" OUP.

Thomas, E. J. & Oman, P. W. (1990) "A bibliography of programming style" ACM SIGPLAN Notices, 25, 2, 7-16.