Beyond Programming:
A Two-Summer Computer Science Institute for Secondary Teachers

Brian Harvey
University of California, Berkeley

Abstract

1992 marks the fourth summer of the Institute for Secondary Mathematics and Computer Science Education (IFSMACSE), an NSF-sponsored program for high school teachers held at Kent State University. The program is divided into separate strands for mathematics and computer science; this paper describes the computer science strand.

In its first three years, IFSMACSE has served about 240 participants, including 90 in computer science. It has, therefore, had a visible impact in Ohio and the neighboring states, especially since one component of the Institute is an Intervening Year Project in which participants bring the ideas of the Institute back to their colleagues through inservice workshops, local curriculum design, and demonstration teaching. (Participants attend the Institute during two consecutive summers.)

IFSMACSE is both technically and pedagogically innovative. Its curriculum goes far beyond what is usually expected of high school teachers (or their students); in computer science, participants study both functional programming (in Logo) and procedural programming (in Pascal) paradigms, and such advanced computer science topics as automata theory and compiler construction. Our high expectations are made realistic by the heavy use of experiential and cooperative learning techniques. Participants learn by experiment, rather than by listening to lectures. In the mathematics strands as well as in computer science, we emphasize the use of programming languages as open-ended experimental media, rather than more specific "courseware."

In addition to describing ways in which IFSMACSE has been successful, this paper discusses two ongoing problems. One is the difficulty that participants have with the idea of functional objects; another is the tenacity with which many participants' abstract educational theories prevent them from observing their own learning experiences clearly.

Beyond Programming:
A Two-Summer Computer Science Institute for Secondary Teachers

The Institute for Secondary Mathematics and Computer Science Education (IFSMACSE) is a summer program for high school teachers that has been funded by the National Science Foundation for three groups of participants over four summers. (Each participant attends the program during two consecutive summers.) It is held at Kent State University, in Kent, Ohio. Participants have been chosen mainly from Ohio and its neighboring states, although a few have come from as far away as Alaska and Pago Pago. The Institute's ambitious and innovative program has attracted a well-known, international faculty that includes high school teachers as well as university professors.

IFSMACSE Directors Ed Dubinsky (Purdue University) and Olaf Stackelberg (Kent State University) began planning for the Institute in 1986 and organized a planning conference in July, 1987. The first group of participants began attending the Institute in the summer of 1989. Here is how one funding proposal described the needs that IFSMACSE was designed to meet:

> **New sophistication in the curriculum.** The traditional subject matter of secondary mathematics has followed a narrow path leading to the calculus. Not only has computer science raised new interest in some classical mathematical topics not currently in the curriculum, but other areas within mathematics (discrete mathematics, abstract algebra, statistics, fractal geometry) have risen from obscurity to new importance. Secondary teachers are often untrained in these areas.
>
> **New methods of teaching.** Organizations such as the National Council of Teachers of Mathematics (NCTM) and the Mathematical Association of America (MAA) have concluded that the traditional approach to pedagogy in mathematics has had too much emphasis on memorization of facts and algorithms, and not enough of the flavor of how mathematics is done and how it is used. IFSMACSE emphasizes learner-centered activities rather than lectures as the primary style of pedagogy.
>
> **New uses of technology.** Computers and graphing calculators are becoming widely available in schools. Teachers require training in these technologies, including general-purpose software such as spreadsheets, software packages designed specifically for education, and new programming languages designed for use in mathematics education.

Each participant attends IFSMACSE for two consecutive summer sessions, each of which is six weeks long. The academic day includes four 90-minute periods. Typically, a participant spends two of those periods in formal class meetings, while the other two periods are available for work in the computer labs. (We use several labs, some containing networked Apple Macintosh computers and the others using networked IBM PS/2 computers.) Evenings are for reading, recreation, additional lab time, and scheduled guest speakers and similar events.

The division of the day between formal class meetings and open lab times may be misleading, because even during formal classes we try to devote as much time as possible to collaborative lab work rather than to lectures or demonstrations. The difference is that during the formal class sessions all members of a class are present at the same time, along with a faculty member. (This ideal of lab-based classwork is not perfectly met. A few of the IFSMACSE classes have relied more heavily on lectures than we'd like, and even in the most experiential class there is an occasional summary lecture. But we try to introduce new concepts through "guided discovery" exercises, saving the lectures for *after* the exploration.)

The participants are divided into mathematics and computer science strands. In the first two IFSMACSE classes, there were two mathematics strands, one focusing on new topics in mathematics, and the other on the use of computer technology in learning mathematics. These were merged for the third class, starting in 1991, by combining courses from the two curricula. The IFSMACSE computer science curriculum has maintained the same general structure for all classes, although of course some of the details have changed.

I was part of the design group for the computer science strand at the 1987 planning meeting, and have taught about half of the courses in that strand at the Institute. In 1990 I was appointed an associate director of IFSMACSE.

The Computer Science Strand

Two overall points of view can be distinguished in computer science education [Harvey, 1991]. The mainstream *software engineering* approach is exemplified by the Computer Science Advanced Placement exam, which emphasizes sequential, procedural programming methodology and top-down design. Courses following this approach are typically taught using Pascal or Modula-2. An alternative view, using *symbolic programming* ideas, was pioneered at the Massachusetts Institute of Technology [Abelson and Sussman, 1985] and emphasizes functional programming style and interactive development. Courses using the second approach are often taught in Scheme or some other dialect of Lisp.

In designing the IFSMACSE computer science curriculum, the participants at the planning meeting wanted to ensure that both approaches would be represented. (Our own personal leanings were mixed.) As a result, Institute participants take two introductory programming courses, one each summer; the first is Logo-based and the second uses Pascal. We chose Logo rather than Scheme to represent the symbolic programming approach partly because the Abelson and Sussman text is more advanced than what we wanted for an introductory high school course; partly because Logo is more generally known in schools; and partly because we had an appropriate Logo-based text available [Harvey, 1985].

Each programming course extends beyond the usual high school level of sophistication, although we think that they are not beyond the abilities of high school students. The Logo course includes very little of the traditional turtle graphics activities, focusing instead on functional programming and list processing. Advanced ideas such as recursion and higher-order functions (that is, functions that operate on other functions as data) are challenging to the participants, but once learned, they allow complicated problems to be solved elegantly and with minimal effort. For example, consider this short program in the Berkeley Logo dialect:

```
TO CHOOSE :MENU [:SOFAR []]
IF EMPTYP :MENU [PRINT :SOFAR STOP]
FOREACH FIRST :MENU [(CHOOSE BUTFIRST :MENU SENTENCE :SOFAR ?)]
END
```

This procedure contains only two instruction lines, but it uses both recursion (the procedure `CHOOSE` is invoked within itself) and the higher-order tool `FOREACH` (applying an instruction to each element of a list) to solve a tree-structured problem. When invoked with a "menu" of independent choices, it prints every possible combination of choices, like this:

```
? CHOOSE [[SMALL MEDIUM LARGE] [VANILLA [RUM RAISIN] GINGER RASPBERRY]
          [CONE CUP]]
SMALL VANILLA CONE
SMALL VANILLA CUP
SMALL RUM RAISIN CONE
SMALL RUM RAISIN CUP
SMALL GINGER CONE
```

```
SMALL GINGER CUP
SMALL RASPBERRY CONE
SMALL RASPBERRY CUP
MEDIUM VANILLA CONE
MEDIUM VANILLA CUP
MEDIUM RUM RAISIN CONE
MEDIUM RUM RAISIN CUP
MEDIUM GINGER CONE
MEDIUM GINGER CUP
MEDIUM RASPBERRY CONE
MEDIUM RASPBERRY CUP
LARGE VANILLA CONE
LARGE VANILLA CUP
LARGE RUM RAISIN CONE
LARGE RUM RAISIN CUP
LARGE GINGER CONE
LARGE GINGER CUP
LARGE RASPBERRY CONE
LARGE RASPBERRY CUP
```

(In dialects of Logo other than Berkeley Logo and Object Logo, the `FOREACH` tool is not automatically provided, but it can be written in Logo.)

The Pascal-based course in the second summer is divided into two three-week components. The first is an introduction to the Pascal language comparable in scope to the Advanced Placement course. The second is a course on algorithms and data structures, roughly like the typical "CS 2" undergraduate course. (For example, last summer's offering used the convex hull problem as its first laboratory assignment. This is the problem of finding which points, out of an arbitrary set of points chosen in the plane, would be touched by a rubber band that was stretched around the entire set and then allowed to contract.) It may seem remarkable to cover the A.P. syllabus in three weeks, but the participants are working full-time on computer science, and they have already had an advanced Logo programming course the previous year. The algorithms course is taught by Selim Akl, a professor at Queen's University in Ontario and the author of three texts in the area of algorithms [Akl, 1985; Akl, 1989; Akl, 1992]. The introduction to Pascal has been taught by Michael Levy, professor at the University of Victoria, and by Darrell Turnidge, an Assistant Dean of Arts and Sciences at Kent State University and an associate director of IFSMACSE.

Each summer's programming course runs in parallel with a second course in which programming is not the focus, although programming may be used as a learning tool. The first summer's course is about the use of computing in schools outside of computer science proper. This course is offered because a high school's computer science teacher is likely also to be the computing expert who is consulted about every aspect of educational computing. This is the course in the IFSMACSE computer science strand that has changed most over the three summers in response to pragmatic needs. As originally designed, the course had two three-week sections. The first was about the technical "nuts and bolts" of using computers: word processing, spreadsheets, and so on, as well as hints about choosing hardware, networking, and similar issues. The second three weeks was an overview of the use of computing throughout the curriculum, combining general readings with a sampler of activities from various disciplines.

This original design has changed for two reasons. The first was that the "nuts and bolts" work turned out to be so popular that participants in the mathematics strands wanted to take the course also. It is taught by Bill Hunt, a Macintosh virtuoso, who is Director of Technology for the Mayfield, Ohio City Schools and a winner of the Presidential Award for Excellence in the Teaching of Mathematics. To make Bill's workshop available to all participants, starting in the second summer it was moved into the evening program.

The second half of the original design was expanded to fill part of the hole thus left in the computer science schedule. The course is taught by Paul Goldenberg, who has taught at all levels from second grade through graduate school and is now a Senior Scientist at Education Development Center in Massachusetts. Paul has enriched the course in two ways. First, he is himself a developer of computer-based curriculum in several areas, including special education [Goldenberg, 1979; Goldenberg, et. al., 1984], linguistics [Goldenberg and Feurzeig, 1987], and fractal geometry [Goldenberg, 1989]. Second, as an experienced process group leader with psychodynamic training, Paul has introduced a group discussion component to the course. He divides the participants into two groups of 15; while one group is working independently in the computer lab, the other is meeting with Paul for a wide-ranging discussion of participants' concerns about educational issues. These discussions have no predetermined agenda, and their character has been very different from one group to another, but they are always strongly connected with other parts of the IFSMACSE program and help participants understand more fully the pedagogic implications of their experiences as learners for their later work as teachers.

The non-programming course in the second summer is a sampler of topics in computer science, including automata theory, discrete mathematics, programming languages, and artificial intelligence [Harvey, 1987]. The rationale for this course is that many students leave high school thinking that a computer scientist is someone who knows two dozen programming languages, and with no idea about the range of topics they'll meet in college. As in IFSMACSE's mathematics courses, the goal is to use the computer as a medium that allows an experimental approach to otherwise abstract ideas. For example, participants explore automata theory using two computer programs, one of which simulates a finite state machine, while the other translates a regular expression into a finite state machine. Before there is any introduction of technical vocabulary, participants use the FSM simulator to play a sort of "guess my rule" game in which the rules specify allowable strings of letters using only the letters A, B, and C. (One possible rule is that the string must start with A and end with C.) FSM notation is introduced as a way to express the game rules in an algorithmic form—one that a computer can use directly to check whether a given string follows a given rule.

Success with Experiential Learning

The Advanced Topics course includes two aspects of programming languages. There is a section about language design, in which technical issues such as scope of variables and call by reference are considered in the context of a comparison between Logo and Pascal. A second section, on compiler construction, is illustrated with a recursive descent compiler for Pascal written in Logo. (The object language, into which a Pascal program is translated, is also Logo. This makes the compiler somewhat unrealistic but also keeps it portable and makes it possible to discuss front-end parsing issues without having to learn about native machine languages.)

I have taught this course twice at IFSMACSE. The first time, I hadn't managed to invent experiments for the participants to use in exploring the ideas. In fact, I didn't think any such activities were possible for the first part, on language design. I did have ideas for activities in the second part, but those ideas were mostly extensions to the already-written Pascal compiler, and required that the participants already understand the structure of the program. Therefore, I ended up doing a lot of lecturing during those weeks. The participants, who had felt quite comfortable with automata theory and discrete mathematics because of the activity-based approach, ended up feeling that the topic of programming languages was too hard for them.

The second summer, before any discussion of programming language design, I gave the participants the following assignment: "Write programs for each of the following in as many languages as possible. Feel free to consult with other people who know these languages better than you do. You should at least be able to do Logo and Pascal; other possibilities are Scheme, ISETL, and BASIC. Some programs may turn out to be impossible in some languages."

1. Mergesort an array or a list of numbers.
2. Shuffle an array or a list of numbers.
3. Exchange the values of two variables.
4. Take a function F as input; output the function that computes $F(F(x))$ for input x.
5. Compute all the prime numbers less than an input N. (Output an array or a list.)
6. Compute all the factors of an input N.

Participants were already familiar with Logo (a list-based language) and Pascal (an array-based language). ISETL is a functional language with a notation designed to be very close to standard mathematical notation, in which the main data aggregate is the *set*, essentially an unordered list. ISETL is not taught in the computer science strand, but is heavily used in the mathematics strand, so the CS participants were encouraged to consult their friends in the other group. Scheme had not previously been used by either strand, but it was installed on the computer network for this week and I used two class sessions to demonstrate it and encourage the participants to play with it.

Participants varied widely in the degree to which they completed the assignment, which called for six programs in each of five languages. But one of the great virtues of the experimental approach is that it isn't necessary for a learner to complete assignments! Any participant who made any serious effort at all quickly saw that only languages with first-class functions, such as Scheme and ISETL, allow the fourth problem to be solved. Most participants—even if their programs never worked—saw that lists are easier to mergesort than arrays, but arrays are easier to shuffle. Filtering problems such as the last two are naturally expressed "all at once" in languages with higher-order functions, but are expressed as iterative loops in languages that emphasize iteration.

A similar activity-first approach also rescued the compiler unit. I had originally chosen to work with a Pascal compiler in order to demonstrate the nature of lexical scope. But the very complexity of this issue, and the corresponding recursive structure of the compiler, made this a bad starting point for an activity approach; the participants would never have written a Pascal compiler from a blank start. Instead, we began the week by taking on the easier task of a BASIC compiler. BASIC is structurally simple because each instruction line begins (not counting the line number) with a keyword (such as `LET` or `PRINT`) that specifies the kind of instruction on the line. The computer lab was arranged as five large clusters of tables, each with about six participants. I assigned each group a statement type, and told them to write a procedure that would take a BASIC instruction line as its input, and would define a Logo procedure to carry out the instruction. We deferred until later the top-level structure that would invoke their procedures and assemble the completed program. (As it turned out, I decided to write this part myself because it involved a few annoying technical issues and no real intellectual ones.)

In a week of work, we never did put together a complete compiler. But the participants all had to work out several issues in their understanding of how to write one. The biggest issue was their tendency to confuse the variables that are part of the compiler itself with variables that belong to the compiled program. This confusion worked both ways; they would try to examine during the compilation process variables that wouldn't exist until the program was run, but they also put into the compiled code references to variables within the compiler. Every group did advance their work to the point at which this confusion was clarified. The issue is a profound one. A compiler breaks down the simplistic wall between program and data in the minds of beginning programmers.

Another issue that came up in the participants' work was tokenization. The tokenization rules for Logo are very similar to those of BASIC; for example, the arithmetic operators are tokens in themselves. But there are a few differences. The most important turned out to be that the comma is a delimiter in BASIC but not in Logo. The groups working on the `INPUT` and `PRINT` statements quickly got bogged down in trying to extract commas from words. I suggested that these groups split their efforts; one subgroup would compile the statements assuming that any comma would be seen as a separate word just as a plus sign would be, while the other subgroup worked only on this tokenization task. Later, in a discussion with the entire class, we all agreed that it would be more sensible to design the compiler in a way that made tokenization an entirely separate task, done uniformly for all statement types, rather than as a subtask of compiling each statement type.

Although we never did examine the Pascal compiler from the textbook in any detail, the participants' experience struggling with compile-time versus run-time variables was enough preparation so that I could tell them about the implications of lexical scoping rules even though they didn't work out the actual process of implementing those rules. Had we had more time, I would have tried to introduce scoping, too, through activities. But the lecture that I gave as a shortcut was vastly more successful than a similar lecture the previous summer to participants who hadn't had the activity of struggling to write a BASIC compiler.

The Difficulty of Understanding Function as Object

The idea of *function* is central to mathematics, and central to the mathematics curriculum at IFSMACSE. Less obviously, it has turned out also to be central to the computer science curriculum.

Students notoriously have trouble understanding functions. For example, this trouble is at the root of their difficulty in freshman calculus. Mathematics educators have focused much research on illuminating functions. It appears that mathematicians understand functions in at least two ways, one of which is much harder than the other [Dubinsky, 1991].

The easier level of understanding is that of *function as process.* This is the view exemplified by the "function machine" of high school algebra, the meat grinder that accepts numbers through a funnel at the top and cranks out other numbers through a chute at the bottom. *Composition of functions* can be understood as hooking the output chute of one machine to the input funnel of another. In this view, the "real things" that we manipulate are the numbers; functions are ways in which we manipulate those things. This view of functions is straightforwardly modeled through procedures in a programming language. (In fact, many languages use the word "function" to name such procedures.) Consider the common metaphor in which a procedure is described as a recipe; it is perfectly clear that the "real things" that we eat are the ingredients, not the recipe itself. You can't eat a recipe, nor can you eat the sequence of steps (turning on the stove, and so on) that the recipe tells us to perform.

The second level requires us to understand *function as object.* In this view, functions are promoted to "real things" that can themselves be manipulated. In calculus, *derivative* is a function whose domain and range are functions. If we represent this function as D, we may loosely write that $D(x^2) = 2x$ but this notation is misleading because x^2 and $2x$ are expressions that represent numbers. We really mean something like

$$D(x \overset{f}{\mapsto} x^2) = (y \overset{g}{\mapsto} 2y)$$

(I have given the function g a different formal parameter to emphasize that this formula is about f and g, not about x or y.) If we want to represent the function D as a machine, it can't be a meat grinder, but rather a "metal grinder" that takes meat grinders—not the values output by meat grinders, but the machines themselves—in its input funnel. Although we are still understanding the *higher-order* function D as a process, we must now understand the functions f and g as objects.

In Logo programming the difficulty about function as object first arises in using the `IF` procedure for conditional execution. In the instruction

```
IF EQUALP :VALUE 3 [PRINT "THREE]
```

why are there brackets around the `PRINT` instruction but not around the `EQUALP` expression? We tell Logo beginners that things within brackets are literal text, while things not in brackets are requests to carry out procedures:

```
? PRINT [SUM 2 3]
SUM 2 3
? PRINT SUM 2 3
5
```

When they meet `IF`, these beginners want to say

```
IF EQUALP :VALUE 3 PRINT "THREE     or
IF [EQUALP :VALUE 3] [PRINT "THREE]
```

Although this presents itself as a syntactic question, about brackets, it's really a problem about functions as objects. `IF` is a higher-order procedure. Its first input is *not* a function, but rather a `TRUE` or `FALSE` value. But its second input *is* a function, in the form of a Logo instruction. The first incorrect form above would invoke `PRINT`, actually printing the word `THREE`, before invoking `IF`. The second incorrect form would put the expression `EQUALP :VALUE 3` in `IF`'s first input funnel, instead of supplying IF with the true or false value that results from invoking EQUALP. Saying the same thing in different words, we are using EQUALP as a function-as-process, but we are using PRINT as a function-as-object.

This difficulty arises in Logo programming because Logo treats IF as a procedure that is syntactically like any other. It requires two inputs, a true-false value and a function. But the idea of a functional object is so hard for many people to learn that some dialects of Logo avoid the issue by making IF a special syntactic construction, unlike everything else in the language, looking more like the IF statements of other languages:

```
IF EQUALP :VALUE 3 THEN PRINT "THREE
```

This gets past the immediate hurdle, but it misses the chance to teach a valuable lesson. Functional objects, once understood, add great expressive power to a language. For example, we can write in Logo a higher-order function called MAP that applies any function to each of the elements of a list:

```
? SHOW MAP "SQUARE [4 5 6 10]
[16 25 36 100]
```

(The FOREACH procedure used in an earlier example is a similar higher-order tool.) MAP performs "all at once" computations that would otherwise require some sort of looping mechanism with auxiliary variables. But why is the function name SQUARE quoted? Ordinarily we quote data words, not procedures. This is exactly the same issue that IF raised; the first input to MAP is the SQUARE function itself, not the result of squaring something. The particular details of punctuation are specific to Logo, but any language that allows higher-order functions needs some notation to distinguish the invocation of a procedure (the use of a function as process) from the procedure itself (function as object).

The same broad issue comes up differently in studying compilers. Here the problem has nothing to do with syntax. It presents itself very directly: We are working with some procedures, the ones in the compiler itself, that we think of purely as process. Other procedures, the ones in the program that is being compiled, must be considered as objects during the compilation process, but then mysteriously represent a process when the compiled program is executed.

Higher-order functions are tools whose usefulness is too great to pass up, and yet our experience with the IFSMACSE participants shows that the underlying ideas are surprisingly difficult to learn. We know some ways to begin; for example, turtle graphics projects can make the issue more concrete [Zazkis and Leron, 1990]. This is an ongoing area of research among the IFSMACSE faculty.

Educational Mythology versus Experience

The final topic in the Advanced Topics course is artificial intelligence. I gave the participants a program called Student to investigate. This program, my translation into Logo of Daniel Bobrow's original 1964 Lisp version, solves algebra word problems by using pattern matching to turn the English sentences into equations. The program, typical of AI projects back then, solves an impressive variety of problems but also fails on an even wider variety. It "understands" certain sentence structures but not others. I had asked the participants to try out problems of their own. If the program failed to solve a problem, I asked them to try to figure out how the program would have to be modified to succeed. In retrospect, this was too hard an assignment, because Student has a very complicated structure and a week was too short a time to allow its exploration.

On the next-to-last day of the Institute, a few of the participants were working on Student, others were doing their Pascal homework, and others were having conversations. At one point a few of the side conversations got quite boisterous, creating an atmosphere in which it was hard for anyone to work. I got angry and let the participants know it, inviting those who weren't working to leave but insisting that they not prevent others from working.

The next day, i.e., the last day of the Institute, Paul Goldenberg invited me to visit his class session with the first-year participants. They had had a discussion about grades, and he was about to hand out a position paper he'd written overnight on the subject. (Of course we don't give grades at IFSMACSE, and the participants all agree that it would ruin the experience for them if we did, but many feel that high school students are different from themselves and need grades.) Paul's paper suggested some ways in which grades can be given to reduce their harmfulness and, in some cases, even allow grades to contribute to the learning process. He thought that I would consider this "a sellout" (his words) and wanted me to present my own views. What I said, in brief, was that dealing with grades is just like any other tension between reform and revolution, and that I think reforms are fine, and even necessary, as long as we design our reforms while keeping in mind the revolution that we *really* want. (I don't expect that sentence to be convincing; the argument about reform and revolution isn't the point I want to make here, but rather part of the background.)

At this point one of the participants said to me, "Don't you think that if you gave grades you wouldn't have had that trouble with the second-year participants yesterday?"

After getting over my (naive) surprise that this story had reached the IFSMACSE gossip mill, I gave two replies. The first was that even if I could have "maintained discipline" by force, which is what she was suggesting, the cost would be too great. I wouldn't trade six weeks of fearless exploration of difficult ideas for one day of decorum. My second reply was a question: "Have you ever taught seniors?"

"Yes."

"Do you give them grades?"

"Yes."

"Do the grades prevent them from having senioritis on the next-to-last day?"

"Are you kidding? They get senioritis in November!"

This participant *knows* that you can't maintain order in a classroom by using grades as a weapon, from her own teaching experience. She *knows* that she herself, at IFSMACSE, works hard and learns rapidly and deeply in an atmosphere without grades, and that if we gave grades she would be more afraid to challenge herself. But she also "knows" that students won't learn anything if you don't force them to. That last bit of knowledge does *not* come from her experience; she learned it from gossip in the staff room, or during her training, or from the newspaper.

In designing IFSMACSE we knew from the beginning that the technical content and the pedagogic content were equally important. Just as we teach the technical ideas not by lecturing, but by participant exploration, we also feel that we can teach our pedagogic ideas best by example. Still, we find that it's important to make explicit, repeatedly, that we are modeling an approach to teaching and learning that will work for the participants and their own students. Many of them find our methods congenial, but many are very reluctant to apply their own observation of their own learning to the situation at home. This fear—a strong word, but appropriate—is another of the difficulties with which we are still struggling at IFSMACSE.

Acknowledgments

The IFSMACSE faculty are all profound thinkers about both education and mathematics, and my work with them has been a terrific intellectual stimulation. The people whose insights are most visible in this paper are Al Cuoco, Ed Dubinsky, Paul Goldenberg, and Uri Leron.

Bibliography

Abelson, Harold, and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

Akl, Selim. *Parallel Sorting Algorithms*, Academic Press, 1985.

Akl, Selim. *The Design and Analysis of Parallel Algorithms*, Prentice Hall, 1989.

Akl, Selim. *Parallel Computational Geometry*, Prentice Hall, 1992.

Dubinsky, Ed. "Constructive Aspects of Reflective Abstraction in Advanced Mathematical Thinking," in L. P. Steffe (ed.), *Epistemological Foundations of Mathematical Experience*, Springer-Verlag, 1991.

Goldenberg, E. Paul. *Special Technology for Special Children*, University Park Press, 1979.

Goldenberg, E. Paul, Susan Jo Russell, Cynthia J. Carter, et al. *Computers, Education and Special Needs*, Addison-Wesley, 1984.

Goldenberg, E. Paul, and Wallace Feurzeig. *Exploring Language with Logo*, MIT Press, 1987.

Goldenberg, E. Paul. "Seeing Beauty in Mathematics: Using Fractal Geometry to Build a Spirit of Mathematical Inquiry," *Journal of Mathematical Behavior*, **8**, 169–204, 1989.

Harvey, Brian. *Computer Science Logo Style, volume 1: Intermediate Programming*, MIT Press, 1985.

Harvey, Brian. *Computer Science Logo Style, volume 3: Advanced Topics*, MIT Press, 1987.

Harvey, Brian. "Symbolic Programming vs. the A.P. Curriculum," *The Computing Teacher*, **18**, 5, February, 1991.

Zazkis, Rina, and Uri Leron. "Implementing Powerful Ideas—The Case of RUN", *Logo Exchange*, **8**, 8, April, 1990.