1969 Turing Award Lecture

Form and Content in Computer Science

MARVIN MINSKY
Massachusetts Institute of Technology
Cambridge, Massachusetts

An excessive preoccupation with formalism is impeding the development of computer science. Form-content confusion is discussed relative to three areas: theory of computation, programming languages, and education.

The trouble with computer science today is an obsessive concern with form instead of content.

No, that is the wrong way to begin. By any previous standard the vitality of computer science is enormous; what other intellectual area ever advanced so far in twenty years? Besides, the theory of computation perhaps encloses, in some way, the science of form, so that the concern is not so badly misplaced. Still, I will argue that an excessive preoccupation with formalism is impeding our development.

Before entering the discussion proper, I want to record the satisfaction my colleagues, students, and I derive from this Turing award. The cluster of questions, once philosophical but now scientific, surrounding the understanding of intelligence was of paramount concern to Alan Turing, and he along with a few other thinkers—notably Warren S. McCulloch and his young associate, Walter Pitts—made many of the

Author's present address: MIT Artificial Intelligence Laboratory; 545 Technology Drive, Cambridge, MA 021390.

early analyses that led both to the computer itself and to the new technology of artificial intelligence. In recognizing this area, this award should focus attention on other work of my own scientific family—especially Ray Solomonoff, Oliver Selfridge, John McCarthy, Allen Newell, Herbert Simon, and Seymour Papert, my closest associates in a decade of work. Papert's views pervade this essay.

This essay has three parts, suggesting form-content confusion in *theory of computation,* in *programming languages,* and in *education.*

1
Theory of Computation

To build a theory, one needs to know a lot about the basic phenomena of the subject matter. We simply do not know enough about these, in the theory of computation, to teach the subject very abstractly. Instead, we ought to teach more about the particular examples we now understand thoroughly, and hope that from this we will be able to guess and prove more general principles. I am not saying this just to be conservative about things probably true that haven't been proved yet. I think that many of our beliefs that seem to be common sense are false. We have bad misconceptions about the possible exchanges between time and memory, trade-offs between time and program complexity, software and hardware, digital and analog circuits, serial and parallel computations, associative and addressed memory, and so on.

It is instructive to consider the analogy with physics, in which one can organize much of the basic knowledge as a collection of rather compact conservation laws. This, of course, is just one kind of description; one could use differential equations, minimum principles, equilibrium laws, etc. Conservation of energy, for example, can be interpreted as defining exchanges between various forms of potential and kinetic energies, such as between height and velocity squared, or between temperature and pressure-volume. One can base a development of quantum theory on a trade-off between certainties of position and momentum, or between time and energy. There is nothing extraordinary about this; any equation with reasonably smooth solutions can be regarded as defining some kind of a trade-off among its variable quantities. But there are many ways to formulate things and it is risky to become too attached to one particular form or law and come to believe that it is the *real* basic principle. See Feynman's [1] dissertation on this.

Nonetheless, the recognition of exchanges is often the conception of a science, if quantifying them is its birth. What do we have, in the computation field, of this character? In the theory of recursive

functions, we have the observation by Shannon [2] that any Turing machine with Q states and R symbols is equivalent to one with 2 states and nQR symbols, and to one with 2 symbols and $n'QR$ states, where n and n' are small numbers. Thus the state-symbol product QR has an almost invariant quality in classifying machines. Unfortunately, one cannot identify the product with a useful measure of machine complexity because this, in turn, has a trade-off with the complexity of the encoding process for the machines—and that trade-off seems too inscrutable for useful application.

Let us consider a more elementary, but still puzzling, trade-off, that between addition and multiplication. How many multiplications does it take to evaluate the 3×3 determinant? If we write out the expansion as six triple-products, we need twelve multiplications. If we collect factors, using the distributive law, this reduces to nine. What is the minimum number, and how does one prove it, in this and in the $n \times n$ case? The important point is not that we need the answer. It is that we do not know how to tell or prove that proposed answers are correct! For a particular formula, one could perhaps use some sort of exhaustive search, but that wouldn't establish a general rule. One of our prime research goals should be to develop methods to prove that particular procedures are computationally minimal, in various senses.

A startling discovery was made about multiplication itself in the thesis of Cook [3], which uses a result of Toom; it is discussed in Knuth [4]. Consider the ordinary algorithm for multiplying decimal numbers: for two n-digit numbers this employs n^2 one-digit products. It is usually supposed that this is minimal. But suppose we write the numbers in two halves, so that the product is $N = (@\,A + B)\,(@\,C + D)$, where @ stands for multiplying by $10^{n/2}$. (The left-shift operation is considered to have negligible cost.) Then one can verify that

$$N = @@AC + BD + @(A + B)\,(C + D) - @(AC + BD).$$

This involves only *three* half-length multiplications, instead of the four that one might suppose were needed. For large n, the reduction can obviously be reapplied over and over to the smaller numbers. The price is a growing number of additions. By compounding this and other ideas, Cook showed that for any ϵ and large enough n, multiplication requires less than $n^{1+\epsilon}$ products, instead of the expected n^2. Similarly, V. Strassen showed recently that to multiply two $m \times m$ *matrices, the number of products could be reduced to the order of* $m^{\log_2 7}$, when it was always believed that the number must be cubic—because there are m^2 terms in the result and each would seem to need a separate inner

product with m multiplications. In both cases ordinary intuition has been wrong for a long time, so wrong that apparently no one looked for better methods. We still do not have a set of proof methods adequate for establishing exactly what is the minimum trade-off exchange, in the matrix case, between multiplying and adding.

The multiply-add exchange may not seem vitally important in itself, but if we cannot thoroughly understand something so simple, we can expect serious trouble with anything more complicated.

Consider another trade-off, that between memory size and computation time. In our book [5], Papert and I have posed a simple question: given an arbitrary collection of n-bit words, how many references to memory are required to tell which of those words is nearest[1] (in number of bits that agree) to an arbitrary given word? Since there are many ways to encode the "library" collection, some using more memory than others, the question stated more precisely is: how must the memory size grow to achieve a given reduction in the number of memory references? This much is trivial: if memory is large enough, only one reference is required, for we can use the question itself as address, and store the answer in the register so addressed. But if the memory is just large enough to store the information in the library, then one has to search all of it—*and we do not know any intermediate results of any value.* It is surely a fundamental theoretical problem of information retrieval, yet no one seems to have any idea about how to set a good lower bound on this basic trade-off.

Another is the serial-parallel exchange. Suppose that we had n computers instead of just one. How much can we speed up what kinds of calculations? For some, we can surely gain a factor of n. But these are rare. For others, we can gain log n, but it is hard to find any or to prove what are their properties. And for most, I think, we can gain hardly anything; this is the case in which there are many highly branched conditionals, so that look-ahead on possible branches will usually be wasted. We know almost nothing about this; most people think, with surely incorrect optimism, that parallelism is usually a profitable way to speed up most computations.

These are just a few of the poorly understood questions about computational trade-offs. There is no space to discuss others, such as the digital-analog question. (Some problems about local versus global computations are outlined in [5].) And we know very little about trades between numerical and symbolic calculations.

There is, in today's computer science curricula, very little attention to what *is* known about such questions; almost all their time is devoted to formal classifications of syntactic language types, defeatist unsolvability theories, folklore about systems programming, and generally trivial fragments of "optimization of logic design"—the latter often in

[1] For identifying an *exact* match, one can use hash-coding and the problem is reasonably well understood.

situations where the art of heuristic programming has far outreached the special-case "theories" so grimly taught and tested—and invocations about programming style almost sure to be outmoded before the student graduates. Even the most seemingly abstract courses on recursive function theory and formal logic seem to ignore the few known useful results on proving facts about compilers or equivalence of programs. Most courses treat the results of work in artificial intelligence, some now fifteen years old, as a peripheral collection of special applications, whereas they in fact represent one of the largest bodies of empirical and theoretical exploration of real computational questions. Until all this preoccupation with form is replaced by attention to the substantial issues in computation, a young student might be well advised to avoid much of the computer science curricula, learn to program, acquire as much mathematics and other science as he can, and study the current literature in artificial intelligence, complexity, and optimization theories.

2 Programming Languages

Even in the field of programming languages and compilers, there is too much concern with form. I say "even" because one might feel that this is one area in which form *ought* to be the chief concern. But let us consider two assertions: (1) languages are getting so they have too much syntax, and (2) languages are being described with too much syntax.

Compilers are not concerned enough with the meanings of expressions, assertions, and descriptions. The use of context-free grammars for describing fragments of languages led to important advances in uniformity, both in specification and in implementation. But although this works well in simple cases, attempts to use it may be retarding development in more complicated areas. There are serious problems in using grammars to describe self-modifying or self-extending languages that involve execution, as well as specifying, processes. One cannot describe syntactically—that is, statically—the valid expressions of a language that is changing. Syntax extension mechanisms must be described, to be sure, but if these are given in terms of a modern pattern-matching language such as SNOBOL, CONVERT [6], or MATCHLESS [7], there need be no distinction between the parsing program and the language description itself. Computer languages of the future will be more concerned with goals and less with procedures specified by the programmer. The following arguments are a little on the extreme side but, in view of today's preoccupation with form, this overstepping will do no harm. (Some of the ideas are due to C. Hewitt and T. Winograd.)

2.1
Syntax
Is Often Unnecessary

One can survive with much less syntax than is generally realized. Much of programming syntax is concerned with suppression of parentheses or with emphasis of scope markers. There are alternatives that have been much underused.

Please do not think I am against the use, at the human interface, of such devices as infixes and operator precedence. They have their place. But their importance to computer science as a whole has been so exaggerated that it is beginning to corrupt the youth.

Consider the familiar algorithm for the square root, as it might be written in a modern algebraic language, ignoring such matters as declarations of data types. One asks for the square root of A, given an initial estimate X and an error limit E.

```
DEFINE SQRT(A,X,E):
    if ABS(A - X * X) < E then X else SQRT(A, (X + A ÷ X) ÷ 2, E).
```

In an imaginary but recognizable version of LISP (see Levin [8] or Weissman [9]), this same procedure might be written:

```
(DEFINE (SQRT AXE)
    (IF (LESS (ABS (-A (* X X))) E) THEN X
    ELSE (SQRT A (÷(+ X (÷A X)) 2) E)))
```

Here, the function names come immediately *inside* their parentheses. The clumsiness, for humans, of writing all the parentheses is evident; the advantages of not having to learn all the conventions, such as that (X+A÷X) is (+X (÷ AX)) and not (÷(+X A) X), is often overlooked.

It remains to be seen whether a syntax with explicit delimiters is reactionary, or whether it is the wave of the future. It has important advantages for editing, interpreting, and for *creation of programs by other programs*. The complete syntax of LISP can be learned in an hour or so; the interpreter is compact and not exceedingly complicated, and students often can answer questions about the system by reading the interpreter program itself. Of course, this will not answer *all* questions about real, practical implementation, but neither would any feasible set of syntax rules. Furthermore, despite the language's clumsiness, many frontier workers consider it to have outstanding expressive power. Nearly all work on procedures that solve problems by building and

modifying hypotheses have been written in this or related languages. Unfortunately, language designers are generally unfamiliar with this area, and tend to dismiss it as a specialized body of "symbol-manipulation techniques."

Much can be done to clarify the structure of expressions in such a "syntax-weak" language by using indentation and other layout devices that are outside the language proper. For example, one can use a "postponement" symbol that belongs to an input preprocessor to rewrite the above as

```
DEFINE (SQRT AXE) ⇓ .
    IF ⇓ THEN X ELSE ⇓ .
     LESS (ABS ⇓ ) E.
                 -A (* X X).
     SQRT A ⇓ E.
            ÷ ⇓ 2.
                +X (÷AX)
```

where the dot means ")(" and the arrow means "insert here the next expression, delimited by a dot, that is available after replacing (recursively) its own arrows." The indentations are optional. This gets a good part of the effect of the usual scope indicators and conventions by two simple devices, both handled trivially by reading programs, and it is easy to edit because subexpressions are usually complete on each line.

To appreciate the power and limitations of the postponement operator, the reader should take his favorite language and his favorite algorithms and see what happens. He will find many choices of what to postpone, and he exercises judgment about what to say first, which arguments to emphasize, and so forth. Of course, ⇓ is not the answer to all problems; one needs a postponement device also for list fragments, and that requires its own delimiter. In any case, these are but steps toward more graphical program-description systems, for we will not forever stay confined to mere strings of symbols.

Another expository device, suggested by Dana Scott, is to have alternative brackets for indicating right-to-left functional composition, so that one can write $\langle\langle\langle x\rangle h\rangle g\rangle f$ instead of $f(g(h(x)))$ when one wants to indicate more naturally what happens to a quantity in the course of a computation. This allows different "accents," as in $f(\langle h(x)\rangle g)$, which can be read: "Compute f of what you get by first computing $h(x)$ and then applying g to it."

The point is better made, perhaps, by analogy than by example. In their fanatic concern with syntax, language designers have become too sentence oriented. With such devices as ⇓ , one can construct objects that are more like paragraphs, without falling all the way back to flow diagrams.

Today's high level programming languages offer little expressive power in the sense of flexibility of style. One cannot control the sequence of presentation of ideas very much without changing the algorithm itself.

2.2
Efficiency and Understanding Programs

What is a compiler for? The usual answers resemble "to translate from one language to another" or "to take a description of an algorithm and assemble it into a program, filling in many small details." For the future, a more ambitious view is required. Most compilers will be systems that "produce an algorithm, given a description of its effect." This is already the case for modern picture-format systems; they do all the creative work, while the user merely supplies examples of the desired formats: here the compilers are more expert than the users. Pattern-matching languages are also good examples. But except for a few such special cases, the compiler designers have made little progress in getting good programs written. Recognition of common subexpressions, optimization of inner loops, allocation of multiple registers, and so forth, lead but to small linear improvements in efficiency—and compilers do little enough about even these. Automatic storage assignments can be worth more. But the real payoff is in analysis of the *computational content* of the algorithm itself, rather than the way the programmer wrote it down. Consider, for example:

DEFINE FIB(N): if $N=1$ then 1, if $N=2$ then 1,
else FIB($N-1$) + FIB($N-2$).

This recursive definition of the Fibonacci numbers 1, 1, 2, 3, 5, 8, 13, · · · can be given to any respectable algorithmic language and will result in the branching tree of evaluation steps shown in Figure 1.

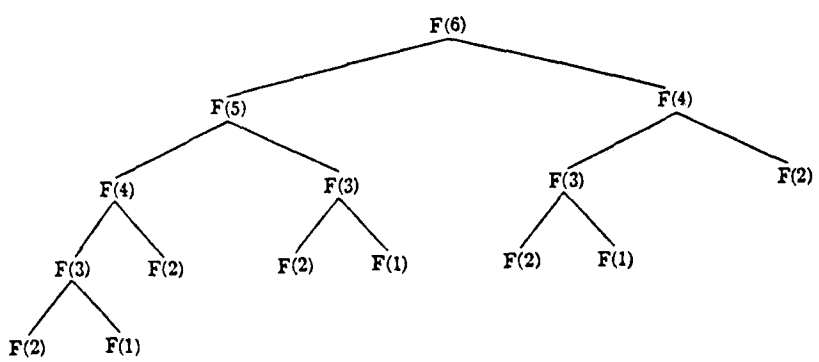

FIGURE 1

One sees that the amount of work the machine will do grows exponentially with N. (More precisely, it passes through the order of FIB(N) evaluations of the definition!) There are better ways to compute this function. Thus we can define two temporary registers and evaluate FIB(N11) in

DEFINE FIB(NAB): if $N=1$ then A else FIB($N-1A+BA$).

which is singly recursive and avoids the branching tree, or even use

```
        A=0
        B=1
LOOP    SWAP AB
        if N=1 return A
        N=N-1
        B=A+B
        goto LOOP
```

Any programmer will soon think of these, once he sees what happens in the branching evaluation. This is a case in which a "course-of values" recursion can be transformed into a simple iteration. Today's compilers don't recognize even simple cases of such transformations, although the reduction in exponential order outweighs any possible gains in local "optimization" of code. It is no use protesting either that such gains are rare or that such matters are the programmer's responsibility. If it is important to save compiling tme, then such abilities could be excised. For programs written in the pattern-matching languages, for example, such simplifications are indeed often made. One usually wins by compiling an efficient tree-parser for BNF system instead of executing brute force analysis-by-synthesis.

To be sure, a systematic theory of such transformations is difficult. A system will have to be pretty smart to detect which transformations are relevant and when it pays to use them. Since the programmer already knows his intent, the problem would often be easier if the proposed algorithm is accompanied (or even replaced) by a suitable goal-declaration expression.

To move in this direction, we need a body of knowledge about analyzing and synthesizing programs. On the theoretical side there is now a lot of activity studying the equivalence of algorithms and schemata, and on proving that procedures have stated properties. On the practical side the works of W. A. Martin [10] and J. Moses [11] illustrate how to make systems that know enough about symbolic transformations of particular mathematical techniques to significantly supplement the applied mathematical abilities of their users.

There is no practical consequence to the fact that the program-reduction problem is recursively unsolvable, in general. In any case one would expect programs eventually to go far beyond human ability in this activity and make use of a large body of program transformations in formally purified forms. These will not be easy to apply directly. Instead, one can expect the development to follow the lines we have seen in symbolic integration, e.g., as in Slagle [12] and Moses [11]. First a set of simple formal transformations that correspond to the elementary entries of a Table of Integrals was developed. On top of these Slagle built a set of heuristic techniques for the algebraic and analytic transformation of a practical problem into those already understood elements; this involved a set of characterization and matching procedures that might be said to use "pattern recognition." In the system of Moses both the matching procedures and the transformations were so refined that, in most practical problems, the heuristic search strategy that played a large part in the performance of Slagle's program became a minor augmentation of the sure knowledge and its skillful application comprised in Moses' system. A heuristic compiler system will eventually need much more *general knowledge* and common sense than did the symbolic integration systems, for its goal is more like making a whole mathematician than a specialized integrator.

2.3 *Describing Programming Systems*

No matter how a language is described, a computer must use a procedure to interpret it. One should remember that *in describing a language the main goal is to explain how to write programs in it and what such programs mean.* The main goal *isn't* to describe the syntax.

Within the static framework of syntax rules, normal forms, Post productions, and other such schemes, one obtains the equivalents of logical systems with axioms, rules of inference, and theorems. To design an unambiguous syntax corresponds then to designing a mathematical system in which each theorem has exactly one proof! But in the computational framework, this is quite beside the point. One has an extra ingredient—control—that lies outside the usual framework of a logical system; an additional set of rules that specify when a rule of inference is to be used. So, for many purposes, ambiguity is a pseudoproblem. If we view a program as a process, we can remember that our most powerful process-describing tools are programs themselves, and *they* are inherently unambiguous.

There is no paradox in defining a programming language by a program. The procedural definition must be understood, of course. One can achieve this understanding by definitions written in another language, one that may be different, more familiar, or simpler than the one being defined. But it is often practical, convenient, and proper

to use the same language! For to understand the definition, one needs to know only the working of that particular program, and not all implications of all possible applications of the language. It is this particularization that makes bootstrapping possible, a point that often puzzles beginners as well as apparent authorities.

Using BNF to describe the formation of expressions may be retarding development of new languages that smoothly incorporate quotation, self-modification, and symbolic manipulation into a traditional algorithmic framework. This, in turn, retards progress toward problem-solving, goal-oriented programming systems. Paradoxically, though modern programming ideas were developed because processes were hard to depict with classical mathematical notations, designers are turning back to an earlier form—the equation—in just the kind of situation that *needs* program. In Section 3, which is on education, a similar situation is seen in teaching, with perhaps more serious consequences.

3
Learning, Teaching, and the "New Mathematics"

Education is another area in which the computer scientist has confused form and content, but this time the confusion concerns his professional role. He perceives his principal function to provide programs and machines for use in old and new educational schemes. Well and good, but I believe he has a more complex responsibility—to work out and communicate models of the process of education itself.

In the discussion below, I sketch briefly the viewpoint (developed with Seymour Papert) from which this belief stems. The following statements are typical of our view:

—To help people learn is to help them build, in their heads, various kinds of computational models.

—This can best be done by a teacher who has, in his head, a reasonable model of what is in the pupil's head.

—For the same reason the student, when debugging his own models and procedures, should have a model of what he is doing, and must know good debugging techniques, such as how to formulate simple but critical test cases.

—It will help the student to know something about computational models and programming. The idea of debugging[2] itself, for example,

[2]Turing was quite good at debugging hardware. He would leave the power on, so as not to lose the "feel" of the thing. Everyone does that today, but it is not the same thing now that the circuits all work on three or five volts.

is a very powerful concept—in contrast to the helplessness promoted by our cultural heritage about gifts, talents, and aptitudes. The latter encourages "I'm not good at this" instead of "How can I make myself better at it?"

These have the sound of common sense, yet they are not among the basic principles of any of the popular educational schemes such as "operant reinforcement," "discovery methods," audio-visual synergism, etc. This is not because educators have ignored the possibility of mental models, but because they simply had no effective way, before the beginning of work on simulation of thought processes, to describe, construct, and test such ideas.

We cannot digress here to answer skeptics who feel it too simpleminded (if not impious, or obscene) to compare minds with programs. We can refer many such critics to Turing's paper [13]. For those who feel that the answer cannot lie in any machine, digital or otherwise, one can argue [14] that machines, when they become intelligent, very likely will feel the same way. For some overviews of this area, see Feigenbaum and Feldman [15] and Minsky [16]; one can keep really up-to-date in this fast-moving field only by reading the contemporary doctoral theses and conference papers on artificial intelligence.

There is a fundamental pragmatic point in favor of our propositions. The child needs models: to understand the city he may use the organism model; it must eat, breathe, excrete, defend itself, etc. Not a very good model, but useful enough. The metabolism of a real organism he can understand, in turn, by comparison with an engine. But to model his own self he *cannot* use the engine or the organism or the city or the telephone switchboard; nothing will serve at all but the computer with its programs and their bugs. Eventually, programming itself will become more important even than mathematics in early education. Nevertheless I have chosen *mathematics* as the subject of the remainder of this paper, partly because we understand it better but mainly because the prejudice against programming as an academic subject would provoke too much resistance. Any other subject could also do, I suppose, but mathematical issues and concepts are the sharpest and least confused by highly charged emotional problems.

3.1
Mathematical Portrait of a Small Child

Imagine a small child of between five and six years, about to enter the first grade. If we extrapolate today's trend, his mathematical education will be conducted by poorly oriented teachers and, partly, by poorly programmed machines; neither will be able to respond to much beyond "correct" and "wrong" answers, let alone to make reasonable inter-

pretations of what the child does or says, because neither will contain good models of the children, or good theories of children's intellectual development. The child will begin with simple arithmetic, set theory, and a little geometry; ten years later he will know a little about the formal theory of the real numbers, a little about linear equations, a little more about geometry, and almost nothing about continuous and limiting processes. He will be an adolescent with little taste for analytical thinking, unable to apply the ten years' experience to understanding his new world.

Let us look more closely at our young child, in a composite picture drawn from the work of Piaget and other observers of the child's mental construction.

Our child will be able to say "one, two, three, . . . " at least up to thirty and probably up to a thousand. He will know the names of some larger numbers but will not be able to see, for example, why ten thousand is a hundred hundred. He will have serious difficulty in counting backwards unless he has recently become very interested in this. (Being good at it would make simple subtraction easier, and might be worth some practice.) He doesn't have much feeling for odd and even.

He can count four to six objects with perfect reliability, but he will not get the same count every time with fifteen scattered objects. He will be annoyed with this, because he is quite sure he should get the same number each time. The observer will therefore think the child has a good idea of the number concept but that he is not too skillful at applying it.

However, important aspects of his concept of number will not be at all secure by adult standards. For example, when the objects are rearranged before his eyes, his impression of their quantity will be affected by the geometric arrangement. Thus he will say that there are fewer x's than y's in:

$$\begin{array}{c} x\ x\ x\ x\ x\ x\ x \\ y\ y\ y\ y\ y\ y\ y \end{array}$$

and when we move the x's to

$$\begin{array}{c} x\ x\ x\ x\ x\ x\ x \\ y\ y\ y\ y\ y\ y\ y \end{array}$$

he will say there are more x's than y's. To be sure, he is answering (in his own mind) a different question about size, quite correctly, but this is exactly the point; the immutability of the number, in such situations, has little grip on him. He cannot use it effectively for reasoning although he shows, on questioning, that he knows that the number of things cannot change simply because they are rearranged. Similarly, when

water is poured from one glass to another (Figure 2(a)), he will say that there is more water in the tall jar than in the squat one. He will have poor estimates about plane areas, so that we will not be able to find a context in which he treats the larger area in Figure 2(b) as four times the size of the smaller one.

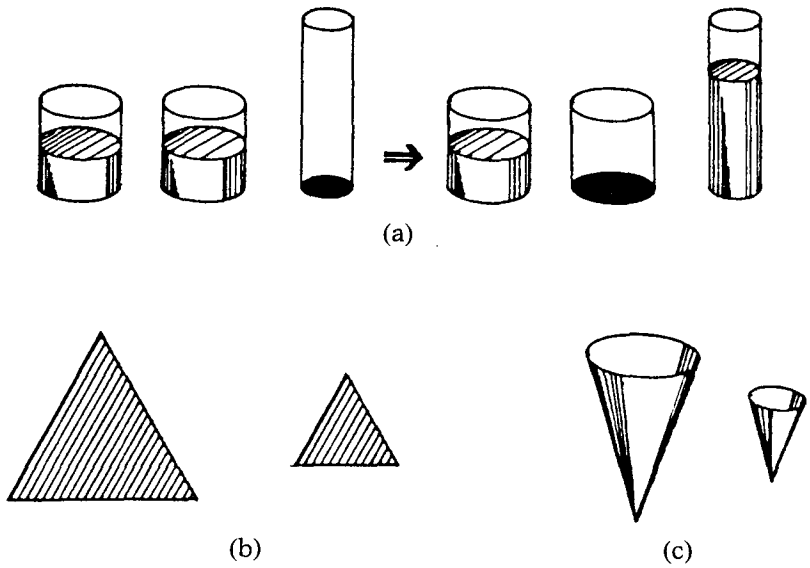

FIGURE 2

When he is an adult, by the way, and is given two vessels, one twice as large as the other, in all dimensions (Figure 2(c)), he will think the one holds about four times as much as the other; probably he will never acquire better estimates of volume.

As for the numbers themselves, we know little of what is in his mind. According to Galton [17], thirty children in a hundred will associate small numbers with definite visual locations in the space in front of their body image, arranged in some idiosyncratic manner such as that shown in Figure 3. They will probably still retain these as adults, and may use them in some obscure semiconscious way to remember telephone numbers; they will probably grow different spatio-visual representations for historical dates, etc. The teachers will never have

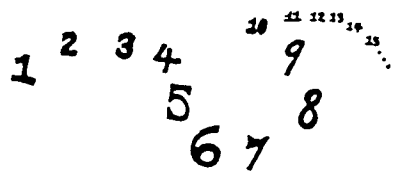

FIGURE 3

heard of such a thing and, if a child speaks of it, even the teacher with her own number form is unlikely to respond with recognition. (My experience is that it takes a series of carefully posed questions before one of these adults will respond, "Oh, yes; 3 is over there, a little farther back.") When our child learns column sums, he may keep track of carries by setting his tongue to certain teeth, or use some other obscure device for temporary memory, and no one will ever know. Perhaps some ways are better than others.

His geometric world is different from ours. He does not see clearly that triangles are rigid, and thus different from other polygons. He does not know that a 100-line approximation to a circle is indistinguishable from a circle unless it is quite large. He does not draw a cube in perspective. He has only recently realized that squares become diamonds when put on their points. The perceptual distinction persists in adults. Thus in Figure 4 we see, as noted by Attneave [18], that the impression of square versus diamond is affected by other alignments in the scene, evidently by determining our choice of which axis of symmetry is to be used in the subjective description.

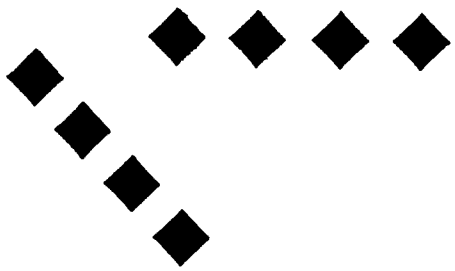

FIGURE 4

Our child understands the topological idea of enclosure quite well. Why? This is a very complicated concept in classical mathematics but in terms of computational processes it is perhaps not so difficult. But our child is almost sure to be muddled about the situation in Figure 5 (see Papert [19]): "When the bus begins its trip around the lake, a boy

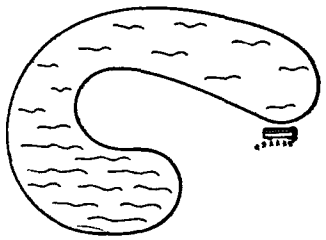

FIGURE 5

is seated on the side away from the water. Will he be on the lake side at some time in the trip?" Difficulty with this is liable to persist through the child's eighth year, and perhaps relates to his difficulties with other abstract double reversals such as in subtracting negative numbers, or with apprehending other consequences of continuity — "At what point in the trip is there any sudden change?" — or with other bridges between logical and global.

Our portrait is drawn in more detail in the literature on developmental psychology. But no one has yet built enough of a computational model of a child to see how these abilities and limitations link together in a structure compatible with (and perhaps consequential to) other things he can do so effectively. Such work is beginning, however, and I expect the next decade to see substantial progress on such models.

If we knew more about these matters, we might be able to help the child. At present we don't even have good diagnostics: his apparent ability to learn to give correct answers to formal questions may show only that he has developed some isolated library routines. If these cannot be called by his central problem-solving programs, because they use incompatible data structures or whatever, we may get a high rated test-passer who will never think very well.

Before computation, the community of ideas about the nature of thought was too feeble to support an effective theory of learning and development. Neither the finite-state models of the Behaviorists, the hydraulic and economic analogies of the Freudians, nor the rabbit-in-the-hat insights of the Gestaltists supplied enough ingredients to understand so intricate a subject. It needs a substrate of already debugged theories and solutions of related but simpler problems. Now we have a flood of such ideas, well defined and implemented, for thinking about thinking; only a fraction are represented in traditional psychology:

symbol table	closed subroutines
pure procedure	pushdown list
time-sharing	interrupt
calling sequence	communication cell
functional argument	common storage
memory protection	decision tree
dispatch table	hardware-software trade-off
error message	serial-parallel trade-off
function-call trace	time-memory trade-off
breakpoint	conditional breakpoint
languages	asynchronous processor
compiler	interpreter
indirect address	garbage collection
macro	list structure
property list	block structure
data type	look-ahead
hash coding	look-behind
microprogram	diagnostic program
format matching	executive program

These are just a few ideas from general systems programming and debugging; we have said nothing about the many more specifically relevant concepts in languages or in artificial intelligence or in computer hardware or other advanced areas. All these serve today as tools of a curious and intricate craft, programming. But just as astronomy succeeded astrology, following Kepler's regularities, the discovery of principles in empirical explorations of intellectual process in machines should lead to a science. (In education we face still the same competition! The *Boston Globe* has an astrology page in its "comics" section. Help fight intellect pollution!)

To return to our child, how can our computational ideas help him with his number concept? As a baby he learned to recognize certain special pair configurations such as two hands or two shoes. Much later he learned about some threes — perhaps the long gap is because the environment doesn't have many fixed triplets: if he happens to find three pennies he will likely lose or gain one soon. Eventually he will find some procedure that manages five or six things, and he will be less at the mercy of finding and losing. But for more than six or seven things, he will remain at the mercy of forgetting; even if his verbal count is flawless, his enumeration procedure will have defects. He will skip some items and count others twice. We can help by proposing better procedures; putting things into a box is nearly foolproof, and so is crossing them off. But for fixed objects he will need some mental grouping procedure.

First one should try to know what the child is doing; eye-motion study might help, asking him might be enough. He may be selecting the next item with some unreliable, nearly random method, with no good way to keep track of what has been counted. We might suggest: *sliding a cursor; inventing easily remembered groups; drawing a coarse mesh.*

In each case the construction can be either real or imaginary. In using the mesh method one has to remember not to count twice objects that cross the mesh lines. The teacher should show that it is good to plan ahead, as in Figure 6, distorting the mesh to avoid the ambiguities!

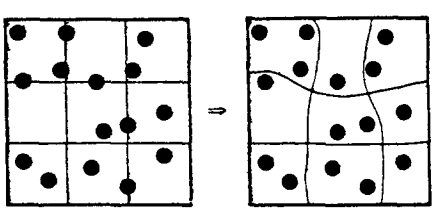

FIGURE 6

Mathematically, the important concept is that "every proper counting procedure yields the same number." The child will understand that any algorithm is proper which (1) *counts all the objects,* (2) *counts none of them twice.*

Perhaps this procedural condition seems too simple; even an adult could understand it. In any case, it is not the concept of number adopted in what is today generally called the "New Math," and taught in our primary schools. The following polemic discusses this.

3.2
The "New Mathematics"

By the "new math" I mean certain primary school attempts to imitate the formalistic outputs of professional mathematicians. Precipitously adopted by many schools in the wake of broad new concerns with early education, I think the approach is generally bad because of form-content displacements of several kinds. These cause problems for the teacher as well as for the child.

Because of the formalistic approach the teacher will not be able to help the child very much with problems of formulation. For she will feel insecure herself as she drills him on such matters as the difference between the empty set and nothing, or the distinction between the "numeral" $3+5$ and the numeral 8 which is the "common name" of the number eight, hoping that he will not ask what is the common name of the fraction $^8/_1$, which is probably different from the rational $^8/_1$ and different from the quotient $^8/_1$ and different from the "indicated division" $^8/_1$ and different from the ordered pair (8,1). She will be reticent about discussing parallel lines. For parallel lines do not usually meet, she knows, but they might (she has heard) if produced far enough, for did not something like that happen once in an experiment by some Russian mathematicians? But enough of the problems of the teacher: let us consider now three classes of objections from the child's standpoint.

Developmental Objections. It is very bad to insist that the child keep his knowledge in a simple ordered hierarchy. In order to retrieve what he needs, he must have a multiply connected network, so that he can try several ways to do each thing. He may not manage to match the first method to the needs of the problem. Emphasis on the "formal proof" is destructive at this stage, because the knowledge needed for finding proofs, and for understanding them, is far more complex (and less useful) than the knowledge mentioned *in* proofs. The network of knowledge one needs for understanding geometry is a web of examples and phenomena, and observations about the similarities and differences between them. One does not find evidence, in children, that such webs are ordered like the axioms and theorems of a logistic system, or that the child could use such a lattice if he had one. *After* one understands a phenomenon, it may be of great value to make a formal system for it, to make it easier to understand more advanced things. But even then, such a formal system is just one of many possible models; the New Math writers seem to confuse their axiom-theorem model with

the number system itself. In the case of the axioms for arithmetic, I will now argue, the formalism is often likely to do more harm than good for the understanding of more advanced things.

Historically, the "set" approach used in New Math comes from a formalist attempt to derive the intuitive properties of the continuum from a nearly finite set theory. They partly succeeded in this stunt (or "hack," as some programmers would put it), but in a manner so complex that one cannot talk seriously about the real numbers until well into high school, if one follows this model. The ideas of topology are deferred until much later. But children in their sixth year already have well-developed geometric and topological ideas, only they have little ability to manipulate abstract symbols and definitions. We should build out from the child's strong points, instead of undermining him by attempting to replace what he has by structures he cannot yet handle. But it is just like mathematicians—certainly the world's worst expositors—to think: "You can teach a child anything, if you just get the definitions precise enough," or "If we get all the definitions right the first time, we won't have any trouble later." We are not programming an empty machine in FORTRAN: we are meddling with a poorly understood large system that, characteristically, uses multiply defined symbols in its normal heuristic behavior.

Intuitive Objections. New Math emphasizes the idea that a number can be identified with an equivalence class of all sets that can be put into one-to-one correspondence with one another. Then the rational numbers are defined as equivalence classes of pairs of integers, and a maze of formalism is introduced *to prevent the child from identifying the rationals with the quotients or fractions.* Functions are often treated as sets, although some texts present "function machines" with a superficially algorithmic flavor. The definition of a "variable" is another fiendish maze of complication involving names, values, expressions, clauses, sentences, numerals, "indicated operations," and so forth. (In fact, there are so many different kinds of data in real problem-solving that real-life mathematicians do *not* usually give them formal distinctions, but use the entire problem context to explain them.) In the course of pursuing this formalistic obsession, the curriculum never presents any coherent picture of real mathematical phenomena of processes, discrete or continuous; of the algebra whose notational syntax concerns it so; or of geometry. The "theorems" that are "proved" from time to time, such as, "A number x has only one additive inverse, $-x$," are so mundane and obvious that neither teacher nor student can make out the purpose of the proof. The "official" proof would add y to both sides of $x + (-y) = 0$, apply the associative law, then the commutative law, then the $y = (-y) = 0$ law, and finally the axioms of equality, to show that y must equal x. The child's mind can more easily understand deeper ideas: "In $x + (-y) = 0$, if y were less than x there would be

some left over; while if x were less than y there would be a minus number left—so they must be exactly equal." The child is not permitted to use this kind of order-plus-continuity thinking, presumably because it uses "more advanced knowledge," hence isn't part of a "real proof." But in the network of ideas the child needs, this link has equal logical status and surely greater heuristic value. For another example, the student is made to distinguish clearly between the *inverse* of addition and the *opposite* sense of distance, a discrimination that seems entirely against the fusion of these notions that would seem desirable.

Computational Objections. The idea of a procedure, and the know-how that comes from learning how to test, modify, and adapt procedures, can transfer to many of the child's other activities. Traditional academic subjects such as algebra and arithmetic have relatively small developmental significance, especially when they are weak in intuitive geometry. (The question of which kinds of learning can "transfer" to other activities is a fundamental one in educational theory: I emphasize again our conjecture that the ideas of procedures and debugging will turn out to be unique in their transferability.) In algebra, as we have noted, the concept of "variable" is complicated; but in computation the child can easily see "$x + y + z$" as describing a procedure (any procedure for adding!) with "x," "y," and "z" as pointing to its "data." Functions are easy to grasp as procedures, hard if imagined as ordered pairs. If you want a graph, describe a machine that draws the graph; if you have a graph, describe a machine that can read it to find the values of the function. Both are easy and useful concepts.

Let us not fall into a cultural trap; the set theory "foundation" for mathematics is popular today among mathematicians because it is the one they tackled and mastered (in college). These scientists simply are not acquainted, generally, with computation or with the Post-Turing-McCulloch-Pitts-McCarthy-Newell-Simon-. . . family of theories that will be so much more important when the children grow up. Set theory is not, as the logicians and publishers would have it, *the* only and true foundation of mathematics; it is a viewpoint that is pretty good for investigating the transfinite, but undistinguished for comprehending the real numbers, and quite substandard for learning about arithmetic, algebra, and geometry.

To summarize my objections, the New Math emphasized the use of formalism and symbolic manipulation instead of the heuristic and intuitive content of the subject matter. The child is expected to learn how to solve problems but we do not teach him what we know, either about the subject or about problem-solving.[3]

[3]In a shrewd but hilarious discussion of New Math textbooks, Feynman [20] explores the consequences of distinguishing between the thing and itself. "Color the picture of the ball red," a book says, instead of "Color the ball red." "Shall we color the entire square area in which the ball image appears or just the part inside the circle of the ball?" asks Feynman. (To "color the *balls* red" would presumably have to be "color the insides of the circles of all the members of the set of balls" or something like that.)

As an example of how the preoccupation with form (in this case, the axioms for arithmetic) can warp one's view of the content, let us examine the weird compulsion to insist that addition is ultimately an operation on just two quantities. In New Math, $a + b + c$ must "really" be one of $(a+(b+c))$ or $((a+b)+c)$, and $a + b + c + d$ can be meaningful only after several applications of the associative law. Now this is silly in many contexts. The child has already a good intuitive idea of what it means to put several sets together; it is just as easy to mix five colors of beads as two. Thus addition is already an n-ary operation. But listen to the book trying to prove that this is not so:

> Addition is . . . always performed on two numbers. This may not seem reasonable at first sight, since you have often added long strings of figures. Try an experiment on yourself. Try to add the numbers 7, 8, 3 simultaneously. No matter how you attempt it, you are forced to choose two of the numbers, add them, and then add the third to their sum.
>
> —From a ninth-grade text

Is the height of a tower the result of adding its stages by pairs in a certain order? Is the length or area of an object produced that way from its parts? Why did they introduce their sets and their one-one correspondences then to miss the point? Evidently, they have talked themselves into believing that the axioms they selected for algebra have some special kind of truth!

Let us consider a few important and pretty ideas that are not discussed much in grade school. First consider the sum $1/2 + 1/4 + 1/8 + \cdots$. Interpreted as area, one gets fascinating regrouping ideas, as in Figure 7. Once the child knows how to do division, he can compute

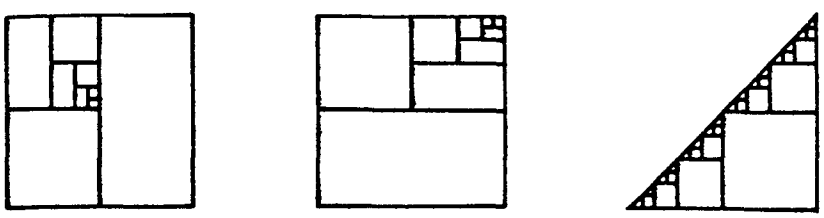

FIGURE 7

and appreciate some quantitative aspects of the limiting process .5, .75, .875, .9375, .96875, $\cdots$, and he can learn about folding and cutting and epidemics and populations. He could learn about $x = px + qx$, where $p + q = 1$, and hence appreciate dilution; he can learn that $3/4, 4/5, 5/6, 6/7, 7/8, \cdots \Rightarrow 1$ and begin to understand the many colorful and common-sense geometrical and topological consequences of such matters.

But in the New Math, the syntactic distinctions between rationals, quotients, and fractions are carried so far that to see which of $3/8$ and $4/9$ is larger, one is not permitted to compute and compare .375 with

.4444. *One must cross-multiply.* Now cross-multiplication is very cute, but it has two bugs: (1) no one can remember which way the resulting conditional should branch, and (2) it doesn't tell how far apart the numbers are. The abstract concept of order is very elegant (another set of axioms for the obvious) but the children already understand order pretty well and want to know the amounts.

Another obsession is the concern for number base. It is good for the children to understand clearly that 223 is "two hundred" plus "twenty" plus "three," and I think that this should be made as simple as possible rather than complicated.[4] I do not think that the idea is so rich that one should drill young children to do arithmetic in several bases! For there is very little transfer of this feeble concept to other things, and it risks a crippling insult to the fragile arithmetic of pupils who, already troubled with $6 + 7 = 13$, now find that $6 + 7 = 15$. Besides, for all the attention to number base, I do not see in my children's books any concern with even a few nontrivial implications—concepts that might justify the attention, such as:

Why is there only one way to write a decimal integer?
Why does casting out nines work? (It isn't even mentioned.)
What happens if we use arbitrary nonpowers, such as $a + 37b + 24c + 11d + \cdots$ instead of the usual $a + 10b + 100c + 1000d + \cdots$?

If they don't discuss such matters, they must have another purpose. My conjecture is that the whole fuss is to make the kids better understand the procedures for multiplying and dividing. But from a developmental viewpoint this may be a serious mistake—in the strategies of both the old and the "new" mathematical curricula. At best, the standard algorithm for long division is cumbersome, and most children will never use it to explore numeric phenomena. And, although it is of some interest to understand how it works, writing out the whole display suggests that the educator believes that the child ought to understand the horrible thing every time! This is wrong. The important idea, if any, is the repeated subtraction; the rest is just a clever but not vital programming hack.

If we can teach, perhaps by rote, a practical division algorithm, fine. But in any case let us give them little calculators; if that is too expensive, why not slide rules. Please, without an impossible explanation. The important thing is to get on to the real numbers! The New Math's concern with integers is so fanatical that it reminds me, if I may mention another pseudoscience, of numerology. (How about that, *Boston Globe!*)

The Cauchy-Dedekind-Russell-Whitehead set-theory formalism was a large accomplishment—another (following Euclid) of a series of demonstrations that many mathematical ideas can be derived from a few primitives, albeit by a long and tortuous route. But the child's

[4] Cf. Tom Lehrer's song, "New Math" [21].

problem is to acquire the ideas at all; he needs to learn about reality. In terms of the concepts available to him, the entire formalism of set theory cannot hold a candle to one older, simpler, and possibly greater idea: the nonterminating decimal representation of the intuitive real number line.

There is a real conflict between the logician's goal and the educator's. The logician wants to minimize the variety of ideas, and doesn't mind a long, thin path. The educator (rightly) wants to make the paths short and doesn't mind—in fact, prefers—connections to many other ideas. And he cares almost not at all about the directions of the links.

As for better understanding of the integers, countless exercises in making little children draw diagrams of one-one correspondences will not help, I think. It *will* help, no doubt, in their learning valuable algorithms, not for number but for the important topological and procedural problems in drawing paths without crossing, and so forth. It is just that sort of problem, now treated entirely accidentally, that we should attend to.

The computer scientist thus has a responsibility to education. Not, as he thinks, because he will have to program the teaching machines. Certainly not because he is a skilled user of "finite mathematics." He knows how to debug programs; he must tell the educators how to help the children to debug their own problem-solving processes. He knows how procedures depend on their data structures; he can tell educators how to prepare children for new ideas. He knows why it is bad to use double-purpose tricks that haunt one later in debugging and enlarging programs. (Thus, one can capture the kids' interest by associating small numbers with arbitrary colors. But what will this trick do for their later attempts to apply number ideas to area, or to volume, or to value?) The computer scientist is the one who must study such matters, because he is the proprietor of the concept of procedure, the secret educators have so long been seeking.

References

1. Feynman, R. P. Development of the space-time view of quantum electrodynamics. *Science 153*, No. 3737 (Aug. 1966), 699–708.
2. Shannon, C. E. A universal Turing machine with two internal states. In *Automata Studies*, Shannon, C. E., and McCarthy, J. (Eds.), Princeton U. Press, Princeton, N.J., 1956, pp. 157–165.
3. Cook, S. A. On the minimum computation time for multiplication. Doctoral diss., Harvard U., Cambridge, Mass., 1966.
4. Knuth, D. *The Art of Computer Programming, Vol. II.* Addison-Wesley, Reading, Mass., 1969.
5. Minsky, M., and Papert, S. *Perceptions: An Introduction to Computational Geometry.* MIT Press, Cambridge, Mass., 1969.
6. Guzman, A., and McIntosh, H. V. CONVERT. *Comm. ACM 9*, 8 (Aug. 1966), 604–615.
7. Hewitt, C. PLANNER: A language for proving theorems in robots. In: Proc. of the International Joint Conference on Artificial Intelligence,

May 7–9, 1969, Washington, D.C., Walker, D.E., and Norton, L M. (Eds.), pp. 295–301.
8. Levin, M., et al. *The LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1965.
9. Weissman, C. *The LISP 1.5 Primer.* Dickenson Pub. Co., Belmont, Calif., 1967.
10. Martin, W. A. Symbolic mathematical laboratory. Doctoral diss., MIT, Cambridge, Mass., Jan. 1967.
11. Moses, J. Symbolic integration. Doctoral diss., MIT, Cambridge, Mass., Dec. 1967.
12. Slagle, J. R. A heuristic program that solves symbolic integration problems in Freshman calculus. In *Computers and Thought,* Feigenbaum, E. A., and Feldman, J. (Eds.), McGraw-Hill, New York, 1963.
13. Turing, A. M. Computing machinery and intelligence. *Mind 59* (Oct. 1950), 433–460; reprinted in *Computers and Thought,* Feigenbaum, E.A., and Feldman, J. (Eds.), McGraw-Hill, New York, 1963.
14. Minsky, M. Matter, mind and models. Proc. IFIP Congress 65, Vol. 1, pp. 45–49 (Spartan Books, Washington, D.C.). Reprinted in *Semantic Information Processing,* Minsky, M. (Ed.), MIT Press, Cambridge, Mass., 1968, pp. 425–432.
15. Feigenbaum, E. A., and Feldman, J. *Computers and Thought.* McGraw-Hill, New York, 1963.
16. Minsky, M. (Ed.). *Semantic Information Processing.* MIT Press, Cambridge, Mass., 1968.
17. Galton, F. *Inquiries into Human Faculty and Development.* Macmillan, New York, 1883.
18. Attneave, F. Triangles as ambiguous figures. *Amer. J. Psychol. 81,* 3 (Sept. 1968), 447–453.
19. Papert, S. Principes analogues à la récurrence. In *Problèmes de la Construction du Nombre,* Presses Universitaires de France, Paris, 1960.
20. Feynman, R. P. New textbooks for the "new" mathematics. *Engineering and Science 28,* 6 (March 1965), 9–15 (California Inst. of Technology, Pasadena).
21. Lehrer, T. New math. In *That Was the Year That Was,* Reprise 6179, Warner Bros. Records.

Categories and Subject Descriptors:

D.3.1 [**Software**]: Formal Definitions and Theory—*syntax*; D.3.4 [**Software**]: Processors—*compilers*; F.2.1 [**Theory of Computation**]: Numerical Algorithms and Problems—*computations on matrices*; F.4.1 [**Theory of Computation**]: Mathematical Logic—*recursive function theory*; I.2.6 [**Computing Methodologies**]: Learning—*concept learning*; K.3.0 [**Computing Milieux**]: Computers and Education—*general*

General Terms:

Algorithms, Languages, Theory

Key Words and Phrases:

Heuristic programming, new math