The 1978 ACM Turing Award was presented to Robert W. Floyd by Walter Carlson, Chairman of the Awards Committee, at the ACM Annual Conference in Washington, D. C., December 4.

In making the selection, the General Technical Achievement Award Subcommittee (formerly the Turing Award Subcommittee) cited Professor Floyd for "helping to found the following important subfields of computer science: the theory of parsing, the semantics of programming languages, automatic program verification, automatic program synthesis, and analysis of algorithms."

Professor Floyd, who received both his A.B. and B.S. from the University of Chicago in 1953 and 1958, respectively, is a self-taught computer scientist. His study of computing began in 1956, when as a night-operator for an IBM 650, he found the time to learn about programming between loads of card hoppers.

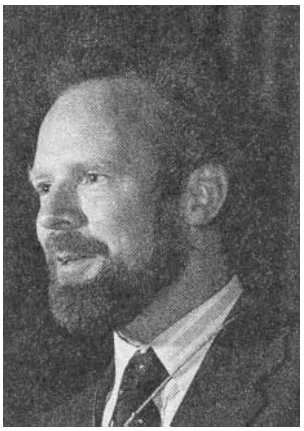
Floyd implemented one of the first Algol 60 compilers, finishing his work on this project in 1962. In the process, he did some early work on compiler optimization. Subsequently, in the years before 1965, Floyd systematized the parsing of programming languages. For that he originated the precedence method, the bounded context method, and the production language method of parsing.

In 1966 Professor Floyd presented a mathematical method to prove the correctness of programs. He has offered, over the years, a number of fast useful algorithms. These include (1) the tree-sort algorithm for in-place sorting, (2) algorithms for finding the shortest paths through networks, and (3) algorithms for finding medians and convex hulls. In addition, Floyd has determined the limiting speed of digital addition and the limiting speeds for permuting information in a computer memory. His contributions to mechanical theorem-proving and automatic spelling checkers have also been numerous.

In recent years Professor Floyd has been working on the design and implementation of a programming language primarily for student use. It will be suitable for teaching structured programming systematically to novices and will be nearly universal in its capabilities.

The Paradigms of Programming

Robert W. Floyd
Stanford University



Paradigm(pæ·radim, −dəim) . . . [a. F. *paradigme*, ad. L. *paradigma*, a. Gr. παραδειγμα pattern, example, f. παραδεικνυ·ναι to exhibit beside, show side by side. . .]
1. A pattern, exemplar, example.
1752 J. Gill *Trinity* v. 91
The archetype, paradigm, exemplar, and idea, according to which all things were made.
From the Oxford English Dictionary.

Today I want to talk about the paradigms of programming, how they affect our success as designers of computer programs, how they should be taught, and how they should be embodied in our programming languages.

A familiar example of a paradigm of programming is the technique of *structured programming*, which appears to be the dominant paradigm in most current treatments of programming methodology. Structured programming, as formulated by Dijkstra [6], Wirth [27, 29], and Parnas [21], among others, consists of two phases.

In the first phase, that of top-down design, or stepwise refinement, the problem is decomposed into a very small number of simpler subproblems. In programming the solution of simultaneous linear equations, say, the first level of decomposition would be into a stage of triangularizing the equations and a following stage of back-substitution in the triangularized system. This gradual decomposition is continued until the subproblems that arise are simple enough to cope with directly. In the simultaneous equation example, the back substitution process would be further decomposed as a backwards iteration of a process which finds and stores the value of the *i*th variable from the *i*th equation. Yet further decomposition would yield a fully detailed algorithm.

The second phase of the structured programming paradigm entails working upward from the concrete objects and functions of the underlying machine to the more abstract objects and functions used throughout the modules produced by the top-down design. In the linear equation example, if the coefficients of the equations are rational functions of one variable, we might first design

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
Author's address: Department of Computer Science, Stanford University, Stanford CA 94305.
© 1979 ACM 0001-0782/79/0800-0455 $00.75.

a multiple-precision arithmetic representation and procedures, then, building upon them, a polynomial representation with its own arithmetic procedures, etc. This approach is referred to as the method of *levels of abstraction*, or of *information hiding*.

The structured programming paradigm is by no means universally accepted. Its firmest advocates would acknowledge that it does not by itself suffice to make all hard problems easy. Other high level paradigms of a more specialized type, such as branch-and-bound [17, 20] or divide-and-conquer [1, 11] techniques, continue to be essential. Yet the paradigm of structured programming does serve to extend one's powers of design, allowing the construction of programs that are too complicated to be designed efficiently and reliably without methodological support.

I believe that the current state of the art of computer programming reflects inadequacies in our stock of paradigms, in our knowledge of existing paradigms, in the way we teach programming paradigms, and in the way our programming languages support, or fail to support, the paradigms of their user communities.

The state of the art of computer programming was recently referred to by Robert Balzer [3] in these words: "It is well known that software is in a depressed state. It is unreliable, delivered late, unresponsive to change, inefficient, and expensive. Furthermore, since it is currently labor intensive, the situation will further deteriorate as demand increases and labor costs rise." If this sounds like the famous "software crisis" of a decade or so ago, the fact that we have been in the same state for ten or fifteen years suggests that "software depression" is a more apt term.

Thomas S. Kuhn, in *The Structure of Scientific Revolutions* [16], has described the scientific revolutions of the past several centuries as arising from changes in the dominant paradigms. Some of Kuhn's observations seem appropriate to our field. Of the scientific textbooks which present the current scientific knowledge to students, Kuhn writes:

> Those texts have, for example, often seemed to imply that the content of science is uniquely exemplified by the observations, laws and theories described in their pages.

In the same way, most texts on computer programming imply that the content of programming is the knowledge of the algorithms and language definitions described in their pages.

Kuhn writes, also:

> The study of paradigms, including many that are far more specialized than those named illustratively above, is what mainly prepares the student for membership in the particular scientific community with which he will later practice. Because he there joins men who learned the bases of their field from the same concrete models, his subsequent practice will seldom evoke overt disagreement over fundamentals...

In computer science, one sees several such communities, each speaking its own language and using its own paradigms. In fact, programming languages typically encourage use of some paradigms and discourage others. There are well defined schools of Lisp programming, APL programming, Algol programming, and so on. Some regard data flow, and some control flow, as the primary structural information about a program. Recursion and iteration, copying and sharing of data structures, call by name and call by value, all have adherents.

Again from Kuhn:

> The older schools gradually disappear. In part their disappearance is caused by their members' conversion to the new paradigm. But there are always some men who cling to one or another of the older views, and they are simply read out of the profession, which thereafter ignores their work.

In computing, there is no mechanism for reading such men out of the profession. I suspect they mainly become managers of software development.

Balzer, in his jeremiad against the state of software construction, went on to prophesy that automatic programming will rescue us. I wish success to automatic programmers, but until they clean the stables, our best hope is to improve our own capabilities. I believe the best chance we have to improve the general practice of programming is to attend to our paradigms.

In the early 1960's, parsing of context-free languages was a problem of pressing importance in both compiler development and natural linguistics. Published algorithms were usually both slow and incorrect. John Cocke, allegedly with very little effort, found a fast and simple algorithm [2], based on a now standard paradigm which is the computational form of dynamic programming [1]. The dynamic programming paradigm solves a problem for given input by first iteratively solving it for all smaller inputs. Cocke's algorithm successively found all parsings of all substrings of the input. In this conceptual frame, the problem became nearly trivial. The resulting algorithm was the first to uniformly run in polynomial time.

At around the same time, after several incorrect top-down parsers had been published, I attacked the problem of designing a correct one by inventing the paradigm of finding a hierarchical organization of processors, akin to a human organization of employers hiring and discharging subordinates, that could solve the problem, and then simulating the behavior of this organization [8]. Simulation of such multiple recursive processes led me to the use of recursive coroutines as a control structure. I later found that other programmers with difficult combinatorial problems, for example Gelernter with his geometry-theorem proving machine [10], had apparently invented the same control structure.

John Cocke's experience and mine illustrate the likelihood that continued advance in programming will require the continuing invention, elaboration, and communication of new paradigms.

An example of the effective elaboration of a paradigm is the work by Shortliffe and Davis on the MYCIN [24] program, which skillfully diagnoses, and recommends medication for, bacterial infections. MYCIN is a

rule-based system, based on a large set of independent rules, each with a testable condition of applicability and a resulting simple action when the condition is satisfied. Davis' TEIRESIAS [5] program modifies MYCIN, allowing an expert user to improve MYCIN's performance. The TEIRESIAS program elaborates the paradigm by tracing responsibility backward from an undesired result through the rules and conditions that permitted it, until an unsatisfactory rule yielding invalid results from valid hypotheses is reached. By this means it has become technically feasible for a medical expert who is not a programmer to improve MYCIN's diagnostic capabilities. While there is nothing in MYCIN which could not have been coded in a traditional branching tree of decisions using conditional transfers, it is the use of the rule-based paradigm, with its subsequent elaboration for self-modification, that makes the interactive improvement of the program possible.

If the advancement of the general art of programming requires the continuing invention and elaboration of paradigms, advancement of the art of the individual programmer requires that he expand his *repertory* of paradigms. In my own experience of designing difficult algorithms, I find a certain technique most helpful in expanding my own capabilities. After solving a challenging problem, I solve it again from scratch, retracing only the *insight* of the earlier solution. I repeat this until the solution is as clear and direct as I can hope for. Then I look for a general rule for attacking similar problems, that *would* have led me to approach the given problem in the most efficient way the first time. Often, such a rule is of permanent value. By looking for such a general rule, I was led from the previously mentioned parsing algorithm based on recursive coroutines to the general method of writing nondeterministic programs [9], which are then transformed by a macroexpansion into conventional deterministic ones. This paradigm later found uses in the apparently unrelated area of problem solving by computers in artificial intelligence, becoming embodied in the programming languages PLANNER [12, 13], MICROPLANNER [25], and QA4 [23].

The acquisition of new paradigms by the individual programmer may be encouraged by reading other people's programs, but this is subject to the limitation that one's associates are likely to have been chosen for their compatibility with the local paradigm set. Evidence for this is the frequency with which our industry advertises, not for programmers, but for Fortran programmers or Cobol programmers. The rules of Fortran can be learned within a few hours; the associated paradigms take much longer, both to learn and to unlearn.

Contact with programming written under alien conventions may help. Visiting MIT on sabbatical this year, I have seen numerous examples of the programming power which Lisp programmers obtain from having a single data structure, which is also used as a uniform syntactic structure for all the functions and operations which appear in programs, with the capability to manipulate programs as data. Although my own previous enthusiasm has been for syntactically rich languages like the Algol family, I now see clearly and concretely the force of Minsky's 1970 Turing lecture [19], in which he argued that Lisp's uniformity of structure and power of self reference gave the programmer capabilities whose content was well worth the sacrifice of visual form. I would like to arrive at some appropriate synthesis of these approaches.

It remains as true now as when I entered the computer field in 1956 that everyone wants to design a new programming language. In the words written on the wall of a Stanford University graduate student office, "I would rather write programs to help me write programs than write programs." In evaluating each year's crop of new programming languages, it is helpful to classify them by the extent to which they permit and encourage the use of effective programming paradigms. When we make our paradigms explicit, we find that there are a vast number of them. Cordell Green [11] finds that the mechanical generation of simple searching and sorting algorithms, such as merge sorting and Quicksort, requires over a hundred rules, most of them probably paradigms familiar to most programmers. Often our programming languages give us no help, or even thwart us, in using even the familiar and low level paradigms. Some examples follow.

Suppose we are simulating the population dynamics of a predator-prey system—wolves and rabbits, perhaps. We have two equations:

$$W' = f(W, R)$$

$$R' = g(W, R)$$

which give the numbers of wolves and rabbits at the end of a time period, as a function of the numbers at the start of the period.

A common beginner's mistake is to write:

```
FOR I := - - - DO
      BEGIN
      W := f(W, R);
      R := g(W, R)
      END
```

where g is, erroneously, evaluated using the modified value of W. To make the program work, we must write:

```
FOR I := - - - DO
      BEGIN
      REAL TEMP;
      TEMP := f(W, R);
      R := g(W, R);
      W := TEMP
      END
```

The beginner is correct to believe we should not have to do this. One of our most common paradigms, as in the predator-prey simulation, is simultaneous assignment of new values to the components of state vectors. Yet hardly any language has an operator for simultaneous assignment. We must instead go through the mechanical, time-wasting, and error-prone operation of introducing

one or more temporary variables and shunting the new values around through them.

Again, take this simple-looking problem:

> Read lines of text, until a completely blank line is found. Eliminate redundant blanks between the words. Print the text, thirty characters to a line, without breaking words between lines.

Because both input and output are naturally expressed using multiple levels of iteration, and because the input iterations do not nest with the output iterations, the problem is surprisingly hard to program in most programming languages [14]. Novices take three or four times as long with it as instructors expect, ending up either with an undisciplined mess or with a homemade control structure using explicit incrementations and conditional execution to simulate some of the desired iterations.

The problem is naturally formulated by decomposition into three communicating coroutines [4], for input, transformation, and output of a character stream. Yet, except for simulation languages, few of our programming languages have a coroutine control structure adequate to allow programming the problem in a natural way.

When a language makes a paradigm convenient, I will say the language *supports* the paradigm. When a language makes a paradigm feasible, but not convenient, I will say the language *weakly supports* the paradigm. As the two previous examples illustrate, most of our languages only weakly support simultaneous assignment, and do not support coroutines at all, although the mechanisms required are much simpler and more useful than, say, those for recursive call-by-name procedures, implemented in the Algol family of languages seventeen years ago.

Even the paradigm of structured programming is at best weakly supported by many of our programming languages. To write down the simultaneous equation solver as one designs it, one should be able to write:

```
MAIN_PROGRAM:
  BEGIN
  TRIANGULARIZE;
  BACK_SUBSTITUTE
  END;
BACK_SUBSTITUTE:
  FOR I := N STEP -1 UNTIL 1 DO
    SOLVE_FOR_VARIABLE(I);
SOLVE_FOR_VARIABLE(I):
  - - -
  - - -
TRIANGULARIZE:
  - - -
  - - -
Procedures for multiple-precision arithmetic
Procedures for rational-function arithmetic
Declarations of arrays
```

In most current languages, one could not present the main program, procedures, and data declarations in this order. Some preliminary human text-shuffling, of a sort readily mechanizable, is usually required. Further, any variables used in more than one of the multiple-precision procedures must be global to every part of the program where multiple-precision arithmetic can be done, thereby allowing accidental modification, contrary to the principle of information hiding. Finally, the detailed breakdown of a problem into a hierarchy of procedures typically results in very inefficient code, even though most of the procedures, being called from only one place, could be efficiently implemented by macroexpansion.

A paradigm at an even higher level of abstraction than the structured programming paradigm is the construction of a hierarchy of languages, where programs in the highest level language operate on the most abstract objects, and are translated into programs on the next lower level language. Examples include the numerous formula-manipulation languages which have been constructed on top of Lisp, Fortran, and other languages. Most of our lower level languages fail to fully support such superstructures. For example, their error diagnostic systems are usually cast in concrete, so that diagnostic messages are intelligible only by reference to the translated program on the lower level.

I believe that the continued advance of programming as a craft requires development and dissemination of languages which support the major paradigms of their user's communities. The design of a language should be preceded by enumeration of those paradigms, including a study of the deficiencies in programming caused by discouragement of unsupported paradigms. I take no satisfaction from the extensions of our languages, such as the variant records and powersets of Pascal [15, 28], so long as the paradigms I have spoken of, and many others, remain unsupported or weakly supported. If there is ever a science of programming language design, it will probably consist largely of matching languages to the design methods they support.

I do not want to imply that support of paradigms is limited to our programming languages proper. The entire environment in which we program, diagnostic systems, file systems, editors, and all, can be analyzed as supporting or failing to support the spectrum of methods for design of programs. There is hope that this is becoming recognized. For example, recent work at IRIA in France and elsewhere has implemented editors which are aware of the structure of the program they edit [7, 18, 26]. Anyone who has tried to do even such a simple task as changing every occurrence of X as an identifier in a program without inadvertently changing all the other X's, will appreciate this.

Now I want to talk about what we *teach* as computer programming. Part of our unfortunate obsession with form over content, which Minsky deplored in his Turing lecture [19], appears in our typical choices of what to teach. If I ask another professor what he teaches in the introductory programming course, whether he answers proudly "Pascal" or diffidently "FORTRAN," I know that he is teaching a grammar, a set of semantic rules, and some finished algorithms, leaving the students to discover, on their own, some process of design. Even the

texts based on the structured programming paradigm, while giving direction at the highest level, what we might call the "story" level of program design, often provide no help at intermediate levels, at what we might call the "paragraph" level.

I believe it is possible to explicitly teach a set of systematic methods for all levels of program design, and that students so trained have a large head start over those conventionally taught entirely by the study of finished programs.

Some examples of what we can teach follow.

When I introduce to students the input capabilities of a programming language, I introduce a standard paradigm for interactive input, in the form of a macro-instruction I call PROMPT_READ_CHECK_ECHO, which reads until the input datum satisfies a test for validity, then echoes it on the output file. This macro is, on one level, itself a paradigm of iteration and input. At the same time, since it reads once more often than it says "Invalid data," it instantiates a more general, previously taught paradigm for the loop executed "*n* and a half times".

```
PROMPT_READ_CHECK_ECHO: arguments are a string
PROMPT, a variable V to be read, and a condition BAD which
characterizes bad data;
PRINT_ON_TERMINAL(PROMPT);
READ_FROM_TERMINAL(V);
WHILE BAD(V) DO
   BEGIN
   PRINT_ON_TERMINAL("Invalid data");
   READ_FROM_TERMINAL(V)
   END;
PRINT_ON_FILE(V)
```

It also, on a higher level, instantiates the responsibilities of the programmer toward the user of the program, including the idea that each component of a program should be protected from input for which that component was not designed.

Howard Shrobe and other members of the Programmer's Apprentice group [22] at MIT have successfully taught their novice students a paradigm of broad utility, which they call generate/filter/accumulate. The students learn to recognize many superficially dissimilar problems as consisting of enumerating the elements of a set, filtering out a subset, and accumulating some function of the elements in the subset. The MACLISP language [18], used by the students, supports the paradigm; the students provide only the generator, the filter, and the accumulator.

The predator-prey simulation I mentioned earlier is also an instance of a general paradigm, the state-machine paradigm. The state-machine paradigm typically involves representing the state of the computation by the values of a set of storage variables. If the state is complex, the transition function requires a design paradigm for handling simultaneous assignment, particularly since most languages only weakly support simultaneous assignment. To illustrate, suppose we want to compute:

$$\frac{\pi}{6} = \arcsin\left(\frac{1}{2}\right) = \boxed{\frac{1}{2}\cdot\boxed{1}} + \boxed{\frac{1}{2^3\cdot 2}\cdot\boxed{3}}$$

$$+ \boxed{\frac{1\cdot 3}{2^5\cdot 2\cdot 4}\cdot\boxed{5}} + \boxed{\frac{1\cdot 3\cdot 5}{2^7\cdot 2\cdot 4\cdot 6}\cdot\boxed{7}} + \cdots$$

where I have circled the parts of each summand that are useful in computing the next one on the right. Without describing the entire design paradigm for such processes, a part of the design of the state transition is systematically to find a way to get from

$$Q = \frac{1\cdot 3}{2^5\cdot 2\cdot 4}, \qquad C = 5$$

$$S = \frac{1}{2} + \frac{1}{2^3\cdot 2\cdot 3} + \frac{1\cdot 3}{2^5\cdot 2\cdot 4\cdot 5}$$

to

$$Q' = \frac{1\cdot 3\cdot 5}{2^7\cdot 2\cdot 4\cdot 6}, \qquad C' = 7$$

$$S' = \frac{1}{2} + \cdots + \frac{1\cdot 3\cdot 5}{2^7\cdot 2\cdot 4\cdot 6\cdot 7}.$$

The experienced programmer has internalized this step, and in all but the most complex cases does it unconsciously. For the novice, seeing the paradigm explicitly enables him to attack state-machine problems more complex than he could without aid, and, more important, encourages him to identify other useful paradigms on his own.

Most of the classical algorithms to be found in texts on computer programming can be viewed as instances of broader paradigms. Simpson's rule is an instance of extrapolation to the limit. Gaussian elimination is problem solution by recursive descent, transformed into iterative form. Merge sorting is an instance of the divide-and-conquer paradigm. For every such classic algorithm, one can ask, "How could I have invented this," and recover what should be an equally classic paradigm.

To sum up, my message to the serious programmer is: spend a part of your working day examining and refining your own methods. Even though programmers are always struggling to meet some future or past deadline, methodological abstraction is a wise long term investment.

To the teacher of programming, even more, I say: identify the paradigms *you* use, as fully as you can, then teach them explicitly. They will serve your students when Fortran has replaced Latin and Sanskrit as the archetypal dead language.

To the designer of programming languages, I say: unless you can support the paradigms I use when I program, or at least support *my* extending your language into one that *does* support my programming methods, I don't need your shiny new languages; like an old car or house, the old language has limitations I have learned to

live with. To persuade me of the merit of your language, you must show me how to construct programs in it. I don't want to discourage the design of new languages; I want to encourage the language designer to become a serious student of the details of the design process.

Thank you, members of the ACM, for naming me to the company of the distinguished men who are my predecessors as Turing lecturers. No one reaches this position without help. I owe debts of gratitude to many, but especially to four men: to Ben Mittman, who early in my career helped and encouraged me to pursue the scientific and scholarly side of my interest in computing; to Herb Simon, our profession's Renaissance man, whose conversation is an education; to the late George Forsythe, who provided me with a paradigm for the teaching of computing; and to my colleague Donald Knuth, who sets a distinguished example of intellectual integrity. I have also been fortunate in having many superb graduate students from whom I think I have learned as much as I have taught them.

To all of you, I am grateful and deeply honored.

Received April 1979

References

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass. 1974.
2. Aho, A.V., and Ullman, J.D. *The Theory of Parsing, Translation, and Compiling, Vol. 1: Parsing.* Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
3. Balzer, R. Imprecise program specification. Report ISI/RR-75-36, Inform. Sciences Inst., Dec. 1975.
4. Conway, M.E. Design of a separable transition-diagram compiler. *Comm. ACM 6*, 7 (July 1963), 396–408.
5. Davis, R. Interactive transfer of expertise: acquisition of new inference rules. Proc. Int. Joint Conf. on Artif. Intell., MIT, Cambridge, Mass., August 1977, pp. 321–328.
6. Dijkstra, E.W. Notes on structured programming. In *Structured Programming*, O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Academic Press, New York, 1972, pp. 1–82.
7. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., and Levy, J.J. A structure oriented program editor: A first step towards computer assisted programming. Res. Rep. 114, IRIA, Paris, April 1975.
8. Floyd, R.W. The syntax of programming languages—a survey. *IEEE EC-13*, 4 (Aug. 1964), 346–353.
9. Floyd, R.W. Nondeterministic algorithms. *J.ACM 14*, 4 (Oct. 1967), 636–644.
10. Gelernter. Realization of a geometry-theorem proving machine. In *Computers and Thought*, E. Feigenbaum and J. Feldman, Eds., McGraw-Hill, New York, 1963, pp. 134–152.
11. Green, C.C., and Barstow, D. On program synthesis knowledge. *Artif. Intell. 10*, 3 (June 1978), 241–279.
12. Hewitt, C. PLANNER: A language for proving theorems in robots. Proc. Int. Joint Conf. on Artif. Intell., Washington, D.C., 1969.
13. Hewitt, C. Description and theoretical analysis (using schemata) of PLANNER... AI TR-258, MIT, Cambridge, Mass., April 1972.
14. Hoare, C.A.R. Communicating sequential processes. *Comm. ACM 21*, 8 (Aug. 1978), 666–677.
15. Jensen, K., and Wirth, N. *Pascal User Manual and Report.* Springer-Verlag, New York, 1978.
16. Kuhn, T.S. *The Structure of Scientific Revolutions.* Univ. of Chicago Press, Chicago, Ill., 1970.
17. Lawler, E., and Wood, D. Branch and bound methods: A survey. *Operations Res. 14*, 4 (July-Aug. 1966), 699–719.
18. MACLISP Manual. MIT, Cambridge, Mass., July 1978.
19. Minsky, M. Form and content in computer science. *Comm. ACM 17*, 2 (April 1970), 197–215.
20. Nilsson, N.J. *Problem Solving Methods in Artificial Intelligence.* McGraw-Hill, New York, 1971.
21. Parnas, D. On the criteria for decomposing systems into modules. *Comm. ACM 15*, 12 (Dec. 1972), 1053–1058.
22. Rich, C., and Shrobe, H. Initial report on a LISP programmer's apprentice. *IEEE J. Software Eng. SE-4*, 6 (Nov. 1978), 456–467.
23. Rulifson, J.F., Derkson, J.A., and Waldinger, R.J. QA4: A procedural calculus for intuitive reasoning. Tech. Note 73, Stanford Res. Inst., Menlo Park, Calif., Nov. 1972.
24. Shortliffe, E.H. *Computer-based Medical Consultations: MYCIN.* American Elsevier, New York, 1976.
25. Sussman, G.J., Winograd, T., and Charniak, C. MICRO-PLANNER reference manual. AI Memo 203A, MIT, Cambridge, Mass., 1972.
26. Teitelman, W., et al. INTERLISP manual. Xerox Palo Alto Res. Ctr., 1974.
27. Wirth, N. Program development by stepwise refinement. *Comm. ACM 14*, (April 1971), 221–227.
28. Wirth, N. The programming language Pascal. *Acta Informatica 1*, 1 (1971), 35–63.
29. Wirth, N. *Systematic Programming, an Introduction.* Prentice-Hall, Englewood Cliffs, New Jersey, 1973.