26-27 July 1976
■ **SIGCSE 76,** Quality Inn/Fort Magruder, Williamsburg, Va. Sponsor: ACM SIGCSE. Chm: Norman Gibbs, Mathematics Department, College of William and Mary, Williamsburg, VA 23185.

9-12 August 1976
Congress of the European Cooperation in Informatics, Amsterdam, The Netherlands. Sponsors: AFCET, AICA, BCS, GI, NTG, NRMG. Contact: M.C. Ashill, 29 Portland Place, London W1N 4HU, England.

10-12 August 1976
■ **1976 ACM Symposium on Symbolic and Algebraic Computation,** IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y. Sponsor: ACM SIGSAM. Chm: James H. Griesmer, IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

10-12 August 1976
■ **Fourth Annual Symposium on the Simulation of Computer Systems,** Boulder, Colorado. Sponsors: ACM, NBS. Gen. chm: Larry G. Hull, Code 533-1, Goddard Space Flight Center, Greenbelt, MD 20771.

8-10 September 1976
The Second International Conference on Very Large Databases, Brussels, Belgium, Cochm.: Erich J. Neuhold, University of Stuttgart; Sakti P. Ghosh, IBM Research Laboratory, San Jose, CA 95193; G.M. Nijssen, Brussels, Belgium,

8-10 September 1976
■ **International Symposium on Technology for Selective Dissemination of Information,** Palazzo dei Congressi Repubblica di S. Marino. Sponsors: ACM Italian Chapter and the Institute of Cybernetics of the Ministry of Education of S. Marino. Contact: Giorgio Valle, Universita di Bologna, Istituto di Elettronica, Viale Risorgimento 2, 40136 Bologna, Italy; 051-582255.

16-17 September 1976
Society for Management Information Systems Annual Conference, Philadelphia, Pa. Contact: SMIS, One First National Plaza, Chicago, IL 60670.

18 September 1976
■ **ACM Mountain Region Computing Conference,** Denver, Colorado. Chm: Shulom Kurtz, K. Inc., P.O. Box 191, Denver, CO 80201; 303 333-9452.

20-24 September 1976
The Second COMPSTAT Symposium on Computational Statistics, Berlin. Contact: COMPSTAT, c/o Universitätsaussenamt der Freien Universität Berlin, Harnackstrasse 4, D-1 Berlin 33, Germany.

22-24 September 1976
■ **APL 76,** Skyline Hotel, Ottawa, Ontario, Canada. Sponsor: ACM SIGPLAN Technical Committee on APL (STAPL). Contact: B.J. Daly, I.P. Sharp Associates Ltd., 2003 Gladstone Ave., Ottawa K2P OY6 Canada.

13-15 October 1976
Second International Conference on Software Engineering, San Francisco, Calif. Sponsor: IEEE-CS. Prog. chm: C. V. Ramamoorthy, Dept. of Electrical Engineering and Computer Sciences, UC Berkeley, Berkeley, CA 94720.

20-22 October 1976
■ **ACM 76 Annual Conference,** Houston, Texas. Chm: Olin Johnson, Computer Science Department, University of Houston, Houston, TX 77004.

8-12 November 1976
■ **Third International Joint Conference on Pattern Recognition,** Hotel Del Coronado. San Diego, Calif. Sponsors: IEEE-CS in cooperation with IEEE Control Systems Society, Information Theory Group, and Systems, Man and Cybernetics Society; ACM; Optical Society of America; Pattern Recognition Society; Society of Photo-Optical Instrumentation Engineers; and IFIP. Chm: Azriel Rosenfeld. Contact: Pattern Recognition, P.O. Box 639, Silver Spring, MD 20901.

27-31 January 1977
American Mathematical Society Annual Meeting, St. Louis, Mo. Contact: American Mathematical Society, P.O. Box 6248, Providence, RI 02940.

31 January-2 February 1977
■ **Computer Science Conference** 77, Atlanta, Ga. Sponsor: ACM. Chm: Vladimir Slamecka, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332.

17-19 March 1977
■ **Tenth Annual Simulation Symposium,** Tampa, Florida. Sponsor: Society for Computer Simulation (SCS) with the cooperation of ACM and IEEE-CS. Contact: Annual Simulation Symposium, P.O. Box 22621, Tampa, FL. 33622.

13-16 June 1977
■ **1977 National Computer Conference,** Dallas Convention Center, Dallas, Tex. Sponsor: AFIPS, 210 Summit Avenue, Montvale, NJ 07645.

17-19 October 1977
■ **ACM 77 Annual Conference,** Olympic Hotel, Washington. Gen. chm: James S. Ketchel, P.O. Box 16156, Seattle, WA 98116; 206 935-6776.

acm forum

Production System Programming

This letter is intended to draw the attention of the Programming Languages community to the notion of production system programming, which has received considerable investigation in the Artificial Intelligence community, and which was recently independently suggested by Edsger Dijkstra. Much can be said about the need for and value of cross fertilization between diverse technical areas: its virtues are commonly agreed to, but it is less commonly practiced, and for quite good reasons. However, cross fertilization can be especially valuable when the diverse areas share a fundamental concern. This is the case now between Programming Languages and Artificial Intelligence. Both areas are vitally concerned with the development of powerful, useful methods of specifying complex information processes. Further, I believe that we are on the verge of a revolution in programming methodology and that production system programming will be a key concept in this revolution. What algebraic languages did for calculation of formulas I expect production system programming and related concepts will do for logical flow of control. In the remainder of this letter I will attempt to show the identity between Dijkstra's "guarded command" and the "production" of Newell et al., and mention some of the uses and characteristics of production system programming.

In his article in the August, 1975, issue of *Communications,* Dijkstra introduces what he calls a "guarded command," and two constructs, the alternative construct and the repetitive construct, for using guarded commands to form executable statements. The guarded command is both syntactically and semantically nearly identical with the "production" defined by Newell and Simon (1972), as will be shown. Productions and production systems as programming devices were discussed by Newell (1967); similar, but not formally identical, programming constructs are credited with much of the success of current applied Artificial Intelligence programs, including MYCIN (Shortliffe, 1973) and DENDRAL (Buchanan et al., 1972).

Syntactic Similarities. Both Dijkstra and Newell and Simon give BNF definitions for their constructs. Dijkstra's BNF follows:

```
⟨guarded command⟩ ::=
        ⟨guard⟩ → ⟨guarded list⟩
⟨guard⟩ ::= ⟨boolean expression⟩
⟨guarded list⟩ ::= ⟨statement⟩
                   .{;⟨statement⟩}
⟨guarded command set⟩ ::=
        ⟨guarded command⟩
        {□⟨guarded command⟩}
⟨alternative construct⟩ ::=
        IF ⟨guarded command set⟩ FI
⟨repetitive construct⟩ ::=
        DO ⟨guarded command set⟩ OD
⟨statement⟩ ::= ⟨alternative-construct⟩ |
                ⟨repetitive construct⟩ |
                "other statements"
```

The relevant portions of the Newell and Simon syntax follow:

```
⟨production-system⟩ ::= ⟨production⟩ |
        ⟨production⟩ ⟨production-system⟩
⟨production⟩ ::= ⟨condition⟩ →
        ⟨action-sequence⟩
⟨action-sequence⟩ ::= ⟨action⟩ |
        ⟨action⟩;⟨action-sequence⟩
⟨action⟩ ::= "other statement"
```

I have used some substitute characters with the obvious meanings, have left out of the Newell and Simon syntax other irrelevant alternatives, and have used "other statement" in the Newell and Simon syntax in place of their actual actions, in the same spirit that Dijkstra used it in his syntax.

The guarded command and the production are clearly identical, as are the guarded command set and the production-system. Newell and Simon have no constructs corresponding to Dijkstra's alternative and repetitive constructs.

Execution. The concept of a production is the same as the concept of a guarded command: the action-

sequence (guarded list) of a production (guarded command) is a sequence of "other statements" that is only eligible for execution when its condition (guard) is satisfied. Just as guarded commands are not themselves executable statements, but must be used within other constructs, the production is also not an executable statement in the usual sense. The production system is an executable construct, and is quite similar to Dijkstra's repetitive construct.

The procedure for executing Dijkstra's repetitive construct is: (1) Select a guarded command G from the guarded command set such that the guard of G evaluates to true; if none, exit. (2) Execute the statements in guarded list of G; go to 1.

Dijkstra specifies that the selection be nondeterministic: more than one guarded command may be eligible, and no rule is given to determine which to select.

The usual interpretation of production-system execution is similar to the above procedure, except that the nondeterminacy is removed by a conflict resolution rule that specifies how to select one of several eligible productions in a production system. The usual rule is that the first eligible production in the production system is selected.

Applications and Advantages. Production systems and similar constructs have been found to have a number of advantages over other programming concepts for the development of knowledge-based programs such as MYCIN and DENDRAL. Some of the advantages experienced are: (1) each production is an intelligible piece of information, or knowledge, whose role and impact on the operation of the system as a whole is relatively easy to understand; (2) the flow of control is governed by the state of the program variables, which are tested by the condition parts of the productions, and no flow charts, branches or GO-TO statements are required; (3) largely as a result of item (2), production system programs tend to be easy to modify, and the effects of modifications are easier to anticipate than in conventional programming practice; (4) production system programs tend to be parsimonious, and (5) there often arises a clean distinction between the logic of the process, which is contained in the productions and their relations to each other, and the predicates and functions used by the productions, which tend to be task-oriented and not involved with the process being programmed.

Production system programming has also revitalized the area of machine learning: Waterman (1974, 1970) has shown that adaptive production systems (i.e. production systems capable of creating new productions and then including them in the existing set of productions) can be successfully applied to rote learning tasks, nonsense syllable association and discrimination tasks, and a serial pattern acquisition task.

Dijkstra uses his constructs in a formal calculus for the derivation of programs, and applies the concepts to correctness proof.

It is very interesting to see essentially the same concept surfacing in so many diverse areas. Another area that might benefit from this concept is structured programming: production system programming strongly encourages top-down program development, and production system programs, when given in a suitable notation, tend to be remarkably clear and concise. In short, it seems that production system programming is an idea whose time has come.

Greg Gibbons
Computer Science Group
Naval Postgraduate School
Monterey, Calif.

REFERENCES

Buchanan, B.G., Feigenbaum, E.A., and Sridharan, N.S. Heuristic theory formation: data interpretation and rule formation. In *Machine Intelligence 7*, Edinburgh U. Press, Edinburgh 1972.

Dijkstra, E.W. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM 18*, 8 (Aug. 75), 453-457.

Newell, A. Studies in problem solving: subject 3 on the cryptarithmetic task: DONALD + GERALD = ROBERT. Carnegie-Mellon U., 1967.

Newell, A., and Simon, H.A. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, N.J., 1972.

Shortliffe, D.A. An artificial intelligence program to advise physicians regarding antimicrobial therapy. *Computers and Biomedical Research 6* (1973), 544-560.

Waterman, D.A. Generalization learning techniques for automating the learning of heuristics. *Artificial Intelligence 1*, 1 and 2 (1970).

Waterman, D.A. Adaptive production systems. Complex Information Processing Working Paper 285, Carnegie-Mellon U., 1974.

Programs as Mobiles, maybe; or Programming as a, well, Plastic Art

Much of the heat in current debate on proper structuring of computer programs appears to be sustained by two combustibles: a realization that programming may be an art, and a conviction that some ways of practicing art are better than others.

These are important persuasions. The interest in them indicates that programmers recognize programs to be more useful as they are more comprehendable, programmers see themselves as creative agents, and they accept the exercise of critical judgment by their fellow artisans. Together these attitudes say that the field is progressive, challenging, and responsible. No one would want to complain about influences which appear to have such desirable effects.

However, as art and criticism move into the profession, they serve as a ready oxidizer for a third influence, the field's habitual identification of programs with written materials. On this identification, consider: the field's technical jargon includes such literary analogs as *words, statements, paragraphs, instructions, lists, indexes, libraries, editors, records, files, documents,* and to get a little farfetched, *symbols* and *characters.* Programs are not "constructed," they are *written,* and they are not written to be "understood" so much as to be *read.*[1] *Language* is just about the only word we have for what it is that programs are made of. The identification is so long-standing and pervasive there can be said to exist within the field almost a confusion of programming with writing. Given the developing interest in art and style, this bias deserves attention. In combination with the concern for art and style it flames especially strong and specific. Specific, that is, in that it inclines programmers to judge programs and program goodness specifically. It inclines the field to consider good programs, not as vaguely artful constructions, but as examples of *good writing.* To put it exactly,

emerging in contemporary programming is a disposition to regard programs as literature, and to invoke in their critical analysis standards appropriate to the literary form.[2]

It may well be that literature *is* an ideal to which programmers should aspire. Literature has at its disposal almost all of the power of natural language. Digital computers can be reasonably viewed as understanding language, or significant variants of language. A piece of paper is a most convenient communication device. But literature is only one art form among several. If the literary form should have characteristics that tend to shape the field, it becomes important to note them and consider their effects. I suggest that the literary form does have influential, and to some degree, unfortunate peculiarities.

(1) The literary form is basically serial.[3] In literature, departures from linearity are risky. For example, how often can an author profitably direct a reader, "Read again the last three pages." It is at least possible that some of the emphasis on blocking in commentary on program structure may be impelled by a desire to see programs set out in the straight lines of literature. A proscription of GOTO would follow as a corollary.

An assumed propriety of the literary form would strongly bias propriety in program structure toward linearity. It would also incline programmers to consider problems well formed only when they had been "straightened out." That is, a literary bias would operate not only in programming but in problem analysis as well.

(2) Literature is, as a form, somewhat Gibralter-like: monolithic and static. Its corresponding virtues are accessibility and permanence; its vices, singularity and immobility. As a form, literature does not gracefully interpret parallelism, interrups, or dynamic constructs such as loops. It tends to make *them* look awkward and suspect. Consequences for programming convention would seem sure to follow.

(3) The genuinely fine idea that programs should be readable has a mischievous side. It inclines the field to accept in the name of good program language design the familiar left-to-right, top-to-bottom reading format.[4] The effect is to insert a conceptual wedge between programs and flow charts, since flow charts are format-incompatible with readable programs. At issue, of course, is whether in programming the division is natural or artificial, to be deepened or bridged.

(4) Since literature is language and symbol oriented, an ideological primacy of literature in the computing field would have the effect of reducing resources available to computing technologies not so based. A practical consequence would be the relative neglect of research and development in analog and hybrid computation.

(5) Literature is so successful as an art form that it has become a professional health hazard. One contractible ailment is complacency: "Our methods are literature-like, therefore elegant and powerful, so go away." Another is the means-ends confusion syndrome: "One more GOTO this week and you're fired."

To consider these things is to admit the complexity of proper programming. Program design is deeply a representational problem. A well designed program embodies a balance of demands issuing from a problem, a computing medium, and a community of users. Keeping the multifaceted nature of programming in mind becomes important in recognizing and dealing with these interface problems.

Another issue pointed to is that the field might find it useful to be more attentive to the sources of its images and more open to images of different kinds. Is the helix a more apt analogy than the loop? If so, what kinds of programming media would foster helical designs? Might it not be useful to ask whether programming media could be made more flexible and problem-conformable.[5] I regularly bless PL/I, but I sometimes wonder if the iterative DO in its present form might be more a relic than a useful tool of thought, and I wonder why I can't have a 3-branch IF in my PL/I program when it's there in the real world. I wonder why I can't sketch a program in three dimensions in much the same way my kids can sketch a building with their toy blocks. And if I *could* rough out a program I would then ask to be able to do it with squirming, wriggly dynamic units if they seemed appropriate to my problem.

There is very likely too much richness in the art and science of computing to allow the field to be satisfied *a priori* with any one art form. If true, the subject itself asks that its graphic, dynamic, architectural, and plastic possibilities be exploited as well as the literary.[6] Such an effort would be aided by a realistic appreciation of the degree to which programming is modeling, and for that reason immeshed in questions of perception, cognition, and social psychology.[7] We do not want to assume that problems are most understandable in hierarchical block form, if they are not. Beneath programming, properly supporting and guiding its development, lie several sciences and possibly *several* arts.

NOTES

[1] Except perhaps for compilers which, by a curious twist of convention, are indeed often referred to as "constructed." Are compilers typically not to be read?

[2] It is possible that literature is not the only model in programming that deserves explication. Mathematics may be another. For example, desirable as it may be to prove program correctness, it remains the fact that proof procedures have their own aims and styles. As Minsky points out in his Turing lecture, at times these may conflict with other important goals. To the extent that programs have a valid expository function, the question naturally arises, might description and proof pull differently in program design?

[3] This is not to say that literature cannot be used to construct complex, nonlinear cognitive structures in the mind—a different aspect of the problem.

[4] The poetic constructions increasingly offered to improve program interpretation are welcome, and as far as they go, helpful. It may be premature to call these constructions poetry, however, because they most often resemble the outlining techniques of prose. Further development may be possible here, but it is instructive that at present such alternatives to the format constraints of prose still appear to be variations of the literary form.

[5] Knuth makes a similar point, for different reasons, in his Turing lecture.

[6] The field is not without examples of sparkling, nonliterary metaphors: *pointers, flags, pictures, images.* Bachman's Turing lecture provides a remarkable allegory: the programmer as a designer of vehicles which move in a problem-space.

[7] Weinberg convincingly illustrates the intersection of programming with behavioral science in *The Psychology of Computer Programming,* (Van Nostrand Reinhold, New York, 1971).

Crayton Walker
Department of Industrial Administration
University of Connecticut
Storrs, CT 06268

Knuth: Reciprocal Table of Inakibit-Anu Incorrectly Rendered

I would like to correct a serious error I made in my paper entitled "Ancient Babylonian Algorithms" [*CACM 15* (July 1972), 671-677].

Near the close of that article I discuss the remarkable reciprocal table of Inakibit-Anu, which appears to be the earliest known example of a large file of data that has been sorted into order. My information was based on Neugebauer's transcription of the tablet AO6456 [3], but unfortunately I failed to read the accompanying German commentary carefully enough, since he departed from his usual custom in this particular case. Many of the lines in his rendition of the table were not on the original clay tablet at all, they were interpolated to show what the table would have looked like if it had been complete. In particular, two lines at the bottom right of p. 675 of my article are due to Neugebauer, not Inakibit: the two lines *not* headed "Reciprocal."

My italicized statement on p. 676, that "this table contains every one" of the 231 regular sexagesimal numbers of six digits or less, is false; the table contains only 136 of those 231. Thus the total size of the file might be estimated at 500 instead of 800.

I still believe that the table continued to two further tablets, now lost. However, the evidence is not conclusive, and E. M. Bruins [1] has argued that there was only one tablet; in his opinion, Inakibit called the table complete, not incomplete. It would be interesting to have a satisfactory explanation for the missing entries, as well as for the mysterious "*O*-numbers" mentioned in the appendix to my paper. (The complete table computed by Gingerich [2] should prove helpful in this regard.) However, the question may be impossible to resolve conclusively since the tablet which survives is only a poor copy of the original.

1 Bruins, E.M. La construction de la grande table de valeurs réciproques *AO* 6456. *Actes de la XVIIe Recontre Assyriologique Internationale*, Comité belge de recherches en Mésopotamie, 1970, *pp.* 99-115.

2 Gingerich, O. Eleven-digit regular sexagesimals and their reciprocals. *Trans. Amer. Phil. Soc. 55*, pt. 8 (1965), 38*pp*.

3 Neugebauer, O. *Mathematische Keilschrift—Texte 1*. Springer-Verlag, Berlin, reprinted 1973, *pp.* 14-22.

Donald E. Knuth
Stanford U., Stanford, CA 94305

AI: McLeod's Position on "Understanding" Questioned

In his letter in the September issue [*Comm. ACM 18*, 9 (Sept. 1975), 546], Mr. McLeod is not, as he fears, belaboring the obvious: on the contrary, he is wide of the mark. He refers to both Colby's and Weizenbaum's programs that imitate forms of human behavior [see *Comm. ACM 9*, 1 (Jan. 1966), 36-45, and *17*, 9 (Sept. 1974), 543] and writes that "... one cannot in general claim that a model is an accurate reflection of reality by examining it in a semantic context which in some sense 'adds' to the model." From this he concludes, in a fashion familiar in Artificial Intelligence, that the model only "appears to understand," and that the human "*interprets* Eliza's responses in a manner which may indeed allow him to conclude that he is being 'understood'. . ." And finally, that such programs are, and can only be, "clever tricks."

I do not want to defend either the methodology, or the detailed performance, of Parry and Eliza here, but I feel it is time someone questioned the sort of reasoning employed by McLeod, which is, alas, all too common.

What, for McLeod, would be a "semantic context" for assessing human-likeness that did *not* "add to the model"—whatever those two quoted phrases may mean? If he means one should not give computer output to someone and ask the loaded question "Did a person or a machine write this?", then he should say so. If he does not mean that, then I cannot see what he could possibly mean. But even if he does mean that, his general point still does not follow, though some detailed argument might make it at least plausible.

One can only use the old game "X appears to understand me, but doesn't really" if one has a clear and independent idea of what it is to understand *really*. We have rules of thumb for everyday life. But, as McLeod knows as well as I, we have no idea about it at all in the sense of psychological or computational models of human understanding that are known to be adequate in a scientific sense. It follows that he cannot know that certain people at certain times do *not* understand in Parry- or Eliza-like ways. That is to say, he has no way of knowing that we do not ourselves sometimes function by means of "clever tricks."

Finally, of course we *interpret* responses "in a manner which may indeed allow (us) to conclude that (we) are being 'understood' . . ." We do it with people, and we do it with machines, because that is what understanding is about, and how could the world be otherwise? The basic flaw in McLeod's position is that, like a lot of people, scientific and lay, he believes that (1) there really is some definitive process or feeling called UNDERSTANDING or BEING-UNDERSTOOD, and (2) that we can know for absolute certainty when we experience it, and (3) we can therefore contrast this feeling with one we have about a machine that "appears" to understand. These assumptions are, alas, false, at least from any scientific point of view, and the fact that Hubert Dreyfus has given a sophisticated philosophical defense [*What Computers Can't Do*, Harper and Row, New York, 1972] of a position very like that of (1)-(3) above, does not make it any more plausible to anyone who believes that the only serious test we can have is how a system *behaves*.

If one sticks to this simple, but firm, principle of machine performance, then McLeod's position will only make sense if and when he can tell us what it would be like to know of any machine that it *really* understood, and didn't just *seem* to do so. I do not believe that this distinction makes much sense, largely because (1)-(3) are false assumptions, yet they are the unexamined foundations of those who argue like Mr. McLeod.

Yorick Wilks
University of Edinburgh
Edinburgh EH8 9NW, Scotland