A Program to Teach Programming

ROBERT R. FENICHEL, JOSEPH WEIZENBAUM, AND JEROME C. YOCHELSON
Massachusetts Institute of Technology, Cambridge, Mass.

The TEACH system was developed at MIT to ease the cost and improve the results of elementary instruction in programming. To the student, TEACH offers loosely guided experience with a conversational language which was designed with teaching in mind. Faculty involvement is minimal.

A term of experience with TEACH is discussed. Pedagogically, the system appears to be successful; straightforward reimplementation will make it economically successful as well.

Similar programs of profound tutorial skill will appear only as the results of extended research. The outlines of this research are beginning to become clear.

KEY WORDS AND PHRASES: elementary programming, computer-assisted learning, UNCL, TEACH
CR CATEGORIES: 1.52

The Problem

A problem confronting the nation's colleges and universities is that of providing instruction in elementary computer programming. An ever-increasing number of academic disciplines are pervaded by computer-based techniques; Computer Science is itself an increasingly popular study. The world of practical affairs outside the university is also making more and more use of the computer, so that the overall demand for people who know how to program digital computer is growing very rapidly. Programmers are a scarce national resource.

The problem is difficult because programming cannot be effectively learned without considerable practice. The student must be given access to a computer. Even more important, the student's work must be coordinated with the lessons to which he is exposed and must somehow be supervised. These requirements, when met in the form of ordinary classroom instruction, generate a large drain on institutional staff resources. Precisely because talented computer people are in extremely short supply, most educational institutions, even those with large financial resources, find it difficult to meet even their self-determined goals.

MIT's large course in elementary computer programming, for example, has been the largest course at the Institute. In the 1967–68 academic year, when there were 950 freshmen, there were 959 students in the large elementary programming course. In addition, several departments offered small elementary courses of their own.

In that same 1967–68 academic year, the large course required at least a part-time commitment from each of 14 faculty and staff members. The salaries of these persons, prorated by the fraction of time devoted to the course, amounted to $53,000. This figure, moreover, takes no account of the fact that at MIT, as at other universities, faculty time is the scarcest of resources.

The course also accounted for $12,000 of rent on punched-card equipment. This equipment, whose utilization was almost entirely chargeable to the course, occupied approximately 1000 square feet of uncharged floor space.

Finally, the course was charged $8,000 for its use of a large-scale computer.

For all this expense, the course has not been effective. Most significant, it has been batch-oriented while MIT has made routine use of time-sharing for five years.

The course has made use of standard batch languages. In the 1967–68 academic year, the language in use was FORTRAN IV. Much of FORTRAN is, in a sense, unteachable. That is, while the language can surely be learned by rote, much of it cannot be logically motivated and explained. Moreover, many important programming concepts are missing in it and in other common batch languages. Surely, one measure of the success of an elementary programming course (at least, one taught in a university context) must be the ease with which students move on to more theoretical courses or to other languages. But such transitions are difficult when the course has never provided experience with such notions as recursion, block structure, functions as objects, or manipulation of strings. Finally, it does not seem possible to teach any of these languages—none of which was designed with teaching in mind—in any pedgogically sound sequence. One cannot teach the transfer-of-control statement before one has taught the use of labels; one cannot teach anything until one has taught the nature of the compile-load-run process; and so on. Without logical sequence, students must take much on faith. This is particularly hard on those students who wish to understand only that core of material which is necessary for their work.

Tools and Goals

It seemed clear to us that, whatever the state of computer-aided instruction (CAI) with respect to other sub-

Work reported herein was supported by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01), and by the National Science Foundation under grant GJ-234.

jects, the computer is the ideal instrument for teaching its own use. If a computer is to be used to teach any laboratory subject, then it will have to be made to simulate that laboratory during some stages of the instruction. This imposes severe difficulties in most cases. But a computer can be made to simulate a computer quite easily. Also, a large part of any laboratory course consists of the student's actual experience with the hardware of the laboratory. In the computer-based teaching system which we have developed, the student is manipulating the laboratory hardware from the very beginning of the course and throughout his instruction.

The major resource we had available to us in the summer of 1967 was the Compatible Time Sharing System (CTSS) [1]. This includes a large number of typewriter consoles, an IBM 7094 computer, and a generous set of software facilities. Of these software facilities, the most important to us were the CTSS file system, a good algebraic language (MAD), a list-processing language (SLIP), and, of course, facilities for typewriter-computer communication.

We began by setting certain objectives. Chief among these was that we intended to teach *programming*, not some particular programming language. We realized, of course, that *some* programming language was required as a vehicle to convey the ideas we intended to communicate. Our task of designing such a language was governed by the following criteria:

(1) A student, having learned our language, should be able to learn any standard algebraic language very quickly, on his own, simply by studying the manual for that language.

(2) Our language should, accordingly, contain all the fundamental ideas of current programming practice. These should be clearly identified and appropriately named—we should not introduce new jargon. Examples of such ideas are: identifiers and variables, control of iteration, recursion, subroutines, and user-defined functions.

(3) Every important idea should be presented only after a need for it has been clearly established. Before an idea is presented to a student, he should be brought to the threshold of inventing the idea himself. For example, he should have had to write a particular program segment several times within a single program before the introduction of the idea of the subroutine. While this criterion is more one of pedagogy than one of language design, it translates into the latter. In particular, it dictates the dependency relationships between the various modules which, when finally joined, constitute the whole language. It determines, for example, that facilities necessary for construction of large program segments must be made available before mechanisms allowing true subroutining. Hence, for example, loop control mechanisms must be independent of subroutine mechanisms.

(4) When a particular facility can be provided in a number of different ways, the design that is capable of being explained most simply and clearly is to be chosen. We believe in general that an implementation that is hard or awkward to explain is probably wrong. The faithful adherence to this design criterion has led us to deep and searching discussions about questions which appeared initially to be simple and even superficial. Our own understanding of language design has deepened and, we believe, the language we have produced is uncommonly clean.[1]

Surrounding the language processor is a teaching system which presents lessons to the student, supervises his progress, and permits him to exercise his skills. Before beginning the detailed design and implementation of this supervisory system, we agreed upon a number of criteria. Among these were:

(1) The individual student's rate of progress through the lesson material is to be governed by the student's own actions. Each student must be given a separate filing area for his work.

(2) The system must remember the individual student's level of progress at all times. System bookkeeping limitations, for example, ought not to make it necessary for the student to repeat or reconstruct work.

(3) The system must detect student errors as quickly as possible, whenever possible in a highly localized context. Once an error is detected, it is to be pointed out to the student in as unambiguous a fashion as possible. Finally, the system must permit the student to correct errors locally, i.e. without forcing him to reconstruct large portions of error-free work.

(4) The system must be totally protected against system failure due to student error.

(5) The system must be modular, so that changes (either in the language to be taught or in the lessons themselves) can be made easily and independently of one another.

System

In accordance with these criteria, the TEACH system was developed to provide virtually instructorless instruction in programming.

At the heart of the TEACH system is a language which we have chosen to call UNCL. We believe that UNCL meets the first set of criteria mentioned above.

UNCL is an interactive language which somewhat resembles Joss, but it differs from Joss and other Joss-like languages in several major respects. A result of these differences (e.g. the presence of block structure, a context editor, and a function-tracing feature) is that UNCL is somewhat more amenable to rational explanation (say, in the metaphors of mathematics), while it is somewhat less amenable to cookbook explanation (the language may not be as appealing to nonprogrammers).

Significantly, UNCL (like Joss) may be used as if it were merely an expensive desk calculator. It is sufficiently rich, on the other hand, that quite complex computational tasks are well within its domain. UNCL is described in somewhat more detail in the Appendix; see also [2].

[1] It will be the subject of a second paper.

Surrounding the UNCL interpreter is another, much simpler interpreter called AGES. Programs written for AGES are called *scripts*. The AGES language is quite general; AGES allows arbitrary recursion, conditional transfers, scanning of student input for keywords, etc. However, the body of scripts which now exists uses little of that generality. This body of scripts constitutes an elementary course in computer programming [3].

The most prominent activity of these scripts is simple typing of information to the student. The material typed is, in sum, a careful text in computer programming. It is divided into 18 *chapters*, each with 5 to 10 *sections*.

The chapters and sections are the calibrations of a ratchet-like mechanism maintained by the scripts. This mechanism allows a student to leave the system and to return days or months later to the point from which he left.

The scripts are able to call upon the UNCL interpreter. Occasionally, the scripts use the interpreter themselves to supply function definitions and other values to the student. More often, the scripts call upon the interpreter only so that the student himself may use it.

For reasons discussed below, the scripts have little control over what a student actually does with the UNCL interpreter. After the scripts have suggested a problem and given the interpreter to him, the student may as well be doing his physics homework as the problem suggested by TEACH.

Whatever else may be said of this freedom for the student, it presents one serious risk. That is, that the student will blunder into use of a feature which he cannot control (e.g. he will blunder into transfers of control before he understands the procedure for interrupting an infinite loop).

To eliminate this risk, the scripts are able to communicate with the syntax scanner of the UNCL interpreter. In particular, the scanner will not recognize any construction which the scripts have not already discussed. In effect, a student is limited to making simple mistakes where the word *simple* is moving as fast as he moves, but no faster.

The scripts engage in limited dialogue with students. Using this capability, the scripts allow a student to request hints about the suggested problems and to determine whether certain sections should be skipped. This feature, along with miscellaneous general purpose features of AGES, also allows the student to review sections which he had previously skipped, or which, for some reason, he may wish to see again.

Experience with Students

A preliminary version of the course was taught in the fall of 1967. The students were about a dozen graduate students and faculty members in the Political Science Department at MIT. Because the scripts were then in crude condition, it was necessary to supplement the students' access to terminals with regular recitation sessions. These sessions provided guidance for the complete reimplementation of scripts and system which was undertaken in the spring of 1968.

Forty new students were exposed to the course in the fall of 1968. Half of these were political scientists similar to the first year's sample; the remainder might as well have been taken at random from the large elementary course: mostly freshman, with some upperclassmen from engineering and scientific departments.

At the beginning of the term, each student was given a short handout telling him how to find a computer terminal around MIT, how to connect it to TEACH, what to do when it runs out of paper, and so on. The terminals were available 112 hours a week, and were used without sign-up lists or other prescheduling. As each student reached the middle of the course (that is, just after he reached the material on transfers of control), he was given a handout on flowcharting techniques. Finally, at the very end of the course, each student was given a long handout describing compiling, batch-processing and various other facts of life with which, once out of the course, he might be faced.

Except for these three handouts, a student's only communication from the instructors came in a personal "mailbox" which was printed for him at the beginning of each session. The instructors used the mailboxes for replies to students' inquiries and for operational announcements of general interest.

One discovery made early in the term was that students would react quite differently to the freedom with which they had been trusted. Some students hoarded their computer time jealously, trying few or none of the suggested problems. Others were fond of extravagant experiment with each possible form of each new mechanism.

Those in the first group were not a new breed. There have always been students who skim in their reading and who turn laboratories into mechanical tests of manual dexterity. There is little to do for such students beyond encouraging them to change their ways.

The spendthrifts of the second group, on the other hand, are quite a different problem. We had to decide what to do with a student who, without having finished the entire body of scripts, had already used up his fair share of computer time.

On the one hand, it seemed unlikely that all of the other students would exhaust their allotments. Although perhaps the one student's excessive usage should have been criticized, perhaps we should not have disconnected him from the system until his usage had *far* exceeded his quota.

We rejected this notion, and we decided to cut off any student overstepping his allotment. We did this because we had come to consider TEACH to be not so much an automated course as an automated book.

That is, we decided to discourage inefficient use of TEACH just as we discourage inefficient use of library books. But when a book is due at the library, it is due, whether or not the borrower has made efficient use of it.

By way of implementing this policy, we arranged for each student, at the end of each session at the terminal, to be informed of the charges assessed against him. He was then responsible for holding his charges to within an announced limit. Each student was informed of the MIT Information Processing Center's rate structure and was encouraged to use ordinary cost-cutting strategies (e.g. to work late at night) on his own behalf. That the units of the meter reading were dollars seems to have lent a felicitous air of reality to these proceedings.

Economic Problems

A prime motivation for development of TEACH was the pressure of a course with a marginal cost of about $75 per student. The marginal cost of the present TEACH is about $150 in machine time and $10 in faculty time per student.

The figure of $150 is, however, an artifact of the present implementation. Of the $150, about $90 is swapping time, the result of having a large, I/O-bound program in CTSS. In a system (such as MULTICS) which allowed shared user procedures, the $90 would be cut substantially; in a dedicated system, the $90 would be cut to zero (of course, some of the marginal cost would then be translated into overhead cost).

Of the remaining $60, we believe that at least $15 would be saved by reimplementation on a machine more adapted to time-sharing than is the 7094.

The only remaining economic issue has to do with the terminal devices. Teletypewriters, at 15 characters per second, are quite adequate for short notes, or for material which requires thoughtful reading.

For reference materials, however, or for discursive description, 15 characters per second is irritatingly slow. Of course, the teletypewriters cannot begin to provide the flowcharts and other graphics to which we should like to expose the student.

One solution to these problems might be to make much more extensive use of handouts, greatly reducing the load on the teletypewriters. Many students have urged such a move, on the grounds that handouts would allow them to give advance thought to problems about to be posed to them.

Other students, however, have favored retention of the endless typing. These students believe that much of their motivation is coupled to the hypnotic rattle of the machine.

Perhaps an optimal solution would involve a combination of handouts and simple graphic terminals. Available graphic terminals, however, are far too expensive for wholesale use with TEACH. Also, students need hard copy of some of their work for later restudy. This portion of their work is small, but it is hard to isolate except in hindsight. Graphic terminals which produce hard copy only on demand (say, via a Polaroid camera) are not adequate; there is a definite gap in this area.

Evaluation

Two full sections of the TEACH course have now been taught. We are thus in a position to evaluate the effectiveness of the course and to make some recommendations about ways in which the research and development effort on the TEACH system should be continued.

The technical aim of completing the system and bringing it into production has certainly been thoroughly met. The TEACH system is smoothly operating. Students almost never report problems in system performance. A large share of the credit for this technical success must go to CTSS, which has once more proved to be a reliable and elegant host for a conversational subsystem.

We have made no tests to determine the efficacy of this teaching method. Our sample was so small that we are doubtful of the utility of any test which could now be performed. Even if our sample were larger, it remains true that in computer programming what is easiest to test (FORTRAN's IJKLMN rule, for example) is of least importance.

The effectiveness of the TEACH scripts will be obscured (in TEACH's favor) by the additional motivation which seems to be inherent in on-line interaction. We do not, however, regard this interaction as only a motivational gimmick. And even if the additional motivation were the only benefit of on-line instruction, arbitrarily to shun this motivation would be to plead for a puritanical, cough-medicine theory of education.

In any event, we lack an independent estimate of the value that the course has been to students. We have, however, held intensive interviews with a fairly large sample of students who had taken this course. These have enabled us to determine that the students in fact absorbed the material which we attempted to teach. We chose students who initially knew little or nothing about programming; now we are impressed with their knowledge. One student, for example, produced a nontrivial FORTRAN IV program over the Christmas holidays, having learned FORTRAN entirely from reading a FORTRAN manual. He reported the transition from UNCL to FORTRAN to have been easy and natural.

Of course, it will undoubtedly require the passage of some time, possibly a year or two, before the effectiveness of TEACH as a first course in programming will be proved or disproved by the performance of its students. But we feel that the course has largely achieved the aims we set for it and for ourselves.

Further Work

Looking ahead, we see problems in both *exporting* TEACH and *enhancing it.*

It seems very clear to us that if TEACH were to be implemented on a machine that is widely available (say, a disk-bearing PDP-10) then it would quickly come to enjoy a rather large public.

Before we make TEACH widely available, however, we must decide what future access students will have to the UNCL language. One of our goals was to teach UNCL in such a way that students will find it very easy to learn other languages (say, ALGOL and FORTRAN) after experience with TEACH. If we had failed to meet this goal, we would have been open to the charge of having taught nothing of intellectual content. But it does not follow that a student brought up on UNCL should be forced to move away from this language as soon as possible.

Given that a TEACH system is about to be created, it is a fairly simple matter to create a somewhat larger system in which the UNCL language can be used independently of TEACH. We can certainly conceive of a system which begins with the course of instruction we have developed, continuing an UNCL interpreter. This system could then go on to permit students to use UNCL independently of the instructional sequence (but still interpretively). Finally, this system could contain facilities for compilation and high speed execution of relatively large programs. We look forward to creation of such a system at MIT.

While the problem of rendering TEACH exportable is largely a production job, the problem of enhancing the current system is an open problem. The fundamental problem is that of checking on what the student is doing.

We have noted that TEACH does not take any notice of what students do with the UNCL interpreter. A good teacher, examining a student's program, often discovers problems of conceptualization of which the student was unaware.

At the moment, we don't know how to program a good teacher. Certainly TEACH could easily be made more obtrusive: it could, for example, easily administer Skinnerian training in its spelling rules. But the interpreter's ordinary diagnostics appear to be quite adequate: we are not convinced that any current CAI techniques would improve TEACH's performance.

In the present system, the student is often asked to perform certain tasks. The system turns control over to him and leaves control with him until he types the word "return." Beyond performing continuous checking for syntactic errors, the system does not oversee what the student actually does. If, for example, we ask the student to write a program to compute square roots, then as soon as he has typed "return" we assume that he has successfully written and tested such a program. Clearly, we should like to have the machine inspect his program from a tutorial point of view. We should like it to be able to help the student, in case he makes *semantic* programming errors as well as purely syntactic ones. This is very hard.

Two questions arise in the context of semantic error correction. One of them is the essentially technical issue of *error detection*. That alone is a substantial problem in artificial intelligence,[2] even if such errors as misleading comments are ignored. For example, a student's algorithm might erroneously be tuned to provide its peak of accuracy and efficiency in special cases which, in fact, do not frequently arise. Another student's procedure might require human intervention in some cases—is this a flaw or a merit?

The other problem of semantic error correction is that of informing a student of what error he has made and offering him *suitable* help at the *appropriate* time. Coming from different students, the same error may be worth different degrees of alarm by the system. In general, the system must be able to go beyond superficially erroneous input to determine whether it was a simple slip or whether it was the result of some deep conceptual misunderstanding.

The system is sufficiently modular that programs for semantic error checking could be inserted without difficulty. The problem, of course, is to write these programs. We now feel that we are beginning to assemble the ideas required for this task.

Our plan now, therefore, is to launch into a fairly substantial research program. We wish to explore these ideas, at first in the context of TEACH. Ultimately we want a system which is a teacher of programming in more than a trivial sense. Our estimate is that useful intermediate results, although not in the form of a production system, will begin to appear within two years of the initiation of the effort.

Acknowledgment. The authors are indebted to Professor S. J. Mason for his thoughtful comments on an earlier version of this paper.

Appendix

The UNCL Interpreter

The UNCL interpreter is highly recursive. At any time during its operation, the current *level* of recursion is a knowable positive integer.

Associated with each recursion level is a scanning pointer (an Instruction Location Counter) within some linear source of statements for execution—a file, the terminal, or a stored program.

The interpreter will return from this level to a lower level if: (1) it executes a RETURN statement; or (2) the statement source runs dry (e.g. end-of-file is encountered); or, (3) it executes a GO TO to a location on the lower level.

The interpreter will save this level and begin execution on the next higher one if: (1) it executes a call to a user-defined function; or (2) it executes a DO statement; or (3) it detects an error in a filed or stored program.

The interpreter accepts *arithmetic* and *Boolean* expressions in various contexts. These have quite ordinary syntax in UNCL, with only a few noteworthy features. (1) Like ALGOL and MAD, but unlike FORTRAN, UNCL has distinct notations for (a) subscription of arrays and (b) application of functions. (2) Like MAD, but unlike ALGOL or FORTRAN, UNCL has distinct notations for (a) functions and (b) applications of no-argument functions.

In several contexts, the interpreter accepts expressions whose values are *character strings* of arbitrary length. The conventions and built-in functions for dealing with strings are more-or-less the same as those of PL/1, except that UNCL performs no type-to-type conversion.

[2] See [6].

A special key on the terminal is an *interrupt button*. A depression of the interrupt button has approximately the effect of an execution error. That is, a message appears and then: (1) if the step being executed was taken from the console, it is simply aborted and lost, whereas (2) if the step being executed was taken from a stored program, it is interrupted and execution on the next higher level is begun, with steps being taken from the console.

Identifiers and statement labels in UNCL may be of arbitrary length.

Statements of UNCL. SET statements (assignment statements) are used in UNCL to assign values to variables or to array elements. Possible right-hand sides include expressions whose values are numbers, truth-values, strings, or functions. Also, the right-hand side may be a *function literal*, as in

```
set factorial = function of x
  local i, f
  set f = 1
  for i = 2 by 1 to x set f = i * f
  return f
end
```

or

```
set factorial = function of n
  if n $e$ 0 return 1
  return n * factorial.(n−1)
end
```

Finally, SET statements must be used in order to create arrays.

TYPE statements are used in UNCL to cause printing at the terminal. A single TYPE statement may specify any number of items for printing. If an item is an expression, it is printed along with a representation of its value. Items which are meta-expressions (e.g. locators of steps in stored programs, file designators, "ALL FUNCTIONS", etc.) cause convenient printing of various sorts. A special item allows examination of the interpreter's Instruction Location Counters for each level of recursion.

The RETURN statement was exemplified in the discussion of SET. RETURN need not carry any value back to the caller.

The DELETE statement in UNCL allows the student to get rid of unwanted variables, program steps, and files. Also, it is possible to delete recursion levels: RETURN will then skip back over deleted levels.

REPLACE statements allow stored programs to be modified. REPLACE statements are often used after an error is detected, while the programs are still in operation.

GO TO statements allow ordinary transfers of control. These transfers may be nonlocal; the recursion level is dropped appropriately.

DEMAND statements allow programs to obtain data from the on-line user.

An IF statement allows conditional execution of an embedded statement, as in the ALGOL construction **if . . . then do . . . .**

FOR statements allow concise descriptions of iteration. The limit, increment, and variable of iteration in FOR are not bound by FOR, so they may freely be modified by the embedded statement. A simple FOR statement was exemplified in the discussion of SET.

DO statements cause the current level of recursion to be saved and execution to begin at the next higher level. Execution continues on this higher level until one of the statements executed is RETURN. There are four varieties of DO, as follows:

(1) DO CONSOLE causes the new level of recursion to watch for steps typed at the console.

(2) DO BLOCK introduces a few steps followed by END; the whole construction has the effect of **begin . . . end** in ALGOL.

(3) DO FILE causes statements to be taken from a named file. The end of the file will serve as a RETURN if one is not encountered before then. If the file name is "CONSOLE" then DO FILE is the same as DO CONSOLE. Similar adjustments are made to other file-handling statements when the file name "CONSOLE" is used with them.

(4) A statement like "DO *f*" is taken to mean "EVALUATE *f*.( )"

LOCAL statements serve to create new generations of variables. These will be available until the current recursion level is relinquished. Any variable not made LOCAL, either explicitly by LOCAL, or implicitly as the formal parameter of a function, is global. Labels, of course, are always local in the sense of ALGOL.

An EVALUATE statement carries an expression to be evaluated. EVALUATE may be used to obtain the side-effects of valueless functions.

An EXIT statement causes UNCL to discontinue its operation and to return control to the host system.

A NULL statement has no effect; neither has a **comment.**

An EDIT statement may be used to expose any stored program to a context editor of the QED variety [4].

An IN FORMAT statement always has within it (a) an embedded TYPE or WRITE statement, and (b) a reference to a string. The string is interpreted as a picture format of the line(s) to be produced by the TYPE or the WRITE.

A READ statement refers (a) to a file name and (b) to a variable or an array element: READ functions like DEMAND, setting the variable or array element to the value of the expression on the next line of the file.

A RESET statement deactivates a named file, so that the next READ of this file will get the first line.

A WRITE statement names a file and is otherwise like TYPE. The lines which TYPE would have produced on-line are appended to the file, which is created if it did not previously exist. Note that if WRITE produces a file α showing the values of various variables, the effect of "DO FILE α" is just to restore the values of the variables. This is especially useful for variables whose values are functions.

TRACE and UNTRACE statements affect the execution of functions named in them. The TRACE/UNTRACE facility closely resembles that of LISP 1.5 [5].

UNCL maintains a count of statements executed between terminal actions. If this count ever exceeds a limit, UNCL detects an interrupt-like error. The limit may be set with a STATEMENT COUNT statement.

The REMARK statement allows students to direct arbitrary remarks to the instructors.

The REVIEW statement allows students to negotiate with AGES for material they might previously have skipped.

Each SHORT statement includes an embedded TYPE or WRITE statement. The effect of the SHORT statement is almost the same as that of the embedded statement. The exception is that lines which would have included an expression and a representation of its value now consist of only the latter. SHORT provides a convenient way to produce files acceptable to READ.

RECEIVED JUNE, 1969: REVISED OCTOBER, 1969

REFERENCES

1. CRISMAN, P. A. (Ed.). *The Compatible Time-Sharing System* (2nd ed.). MIT Press, Cambridge, Mass., 1965.
2. FENICHEL, ROBERT R., AND YOCHELSON, JEROME C. PL/2 reference manual. Project MAC M-388 (unpublished), 1969.
3. FENICHEL, ROBERT R. The Teach System: Scripts for a program to teach programming. Unpublished, 1969.
4. DEUTSCH, L. PETER, AND LAMPSON, BUTLER W. An online editor. *Comm. ACM 10*, 12 (Dec., 1967), 793–799, 803.
5. MCCARTHY, JOHN, et al. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1965.
6. MINSKY, MARVIN L. ACM 1969 A.M. Turing Lecture. *J. ACM* (to appear).