A STRUCTURED APL APPROACH TO COMPUTER AIDED INSTRUCTION

Wilbur R. LePage
Department of Electrical and Computer Engineering
Syracuse University
Syracuse, N.Y. 13210
315-423-4377

Abstract

The APL-based system for Computer Aided Instruction described in this paper is presented as a general purpose system having interest in its own right, and also as an illustration of a "structured" approach to the solution of a relatively complex computing problem. The system meets the more or less mandatory requirement that neither the student user nor the teacher who creates the CAI instructional material should be required to know APL; but the main features are the ease with which a CAI lesson can be created, edited, and modified by a teacher, the "transparency" of the APL system with respect to a student user, and a "modular" design which permits modifications of the system to be made easily. A student employs only one APL function, and there are four functions used in writing the text. Also, in files there are related sets of permanent control functions, and functions and variables related to the text material, which exist in the active workspace only while the system is in use. The system includes a capability for monitoring the progress of students.

Introduction

The work described in this paper evolved as a result of the need for supplementary instruction in an APL-based problem solving course in engineering. Because of the problem solving orientation, and the reasonably high level of APL involved, there was no desire to use CAI for the entire course. However, the delegation of certain aspects to CAI, such as workspace management and function editing, seemed to be a time saving possibility. It was further recognized that the system should be easy to use and highly flexible, that the teacher would not have much time for "writing" lessons, and that with the gaining of experience there would be frequent occasions for revising the CAI material. This context is mentioned with the thought that there might be similar applications in other subject areas.

Many of the principles incorporated in this system have been recognized previously[1] but some were difficult or impossible to attain on earlier APL systems. The realization described here is on an I.P. Sharp system and employs files, error trapping, and packages. These make it possible for a student's active workspace to remain free of all objects other than the one function employed to initiate a lesson, except during execution of that function, and for APL error reports to be suppressed. A file capability is an absolutely minimum requirement.

A considerable amount of sophisticated programming is involved, particularly in the function for checking student answers according to various criteria, but for the most part standard idioms are used, and so the details of the functions are omitted. Emphasis is on the design of the system and specifications of required properties of the functions.

Overall Design Criteria

The design criteria outlined below stem from a consideration of the process of learning, modified by compromises which are necessary to keep the complexity of the system within reasonable bounds.

(a) Very little knowledge of APL should be required of the student and the teacher who writes the CAI text material. For the student the only requirement should be an ability to sign on, load a workspace, and execute one prepared function. The teacher should be required only to be able to use a few prepared functions, and catenation, for the purpose of composing and editing the text material.

(b) The CAI programs should provide optional branching to remedial material in the event of a wrong answer (with an optional remedial question), but such branching should not be selective in accordance with the nature of a wrong answer. Furthermore, the teacher should not be required to write explicit branching statements.

(c) The CAI text material should be organized into groups of "lessons," each having an appropriate length for one learning session. Furthermore, particularly for the convenience of the person who writes the text, each lesson should be composed of a set of numbered "sections" to be executed sequentially by a student. In general, a section should deal with closely related topics, but with the possibility of including more than one statement of information and question, and optional remedial material associated with each question. It should be possible for the teacher to schedule an interruption of the normal sequencing of sections, for example, at a point where a student is expected to do some original experimenting on the computer.

(d) There should be a facility for revising individual sections, removing them completely, or adding new ones without disturbing the others.

(e) A simple method should be provided whereby the writer of the text material can specify how student answers should be compared with predetermined standard answers (check for complete identity, search for one or more keywords, ignore punctuation marks and extra spaces or not, etc.).

(f) A student should be able to use the system either by loading his or her own library workspace (which will contain a previously copied function) or by loading a workspace from another number. Provision should be made for the automatic reloading of the pertinent workspace whenever there is a return to the immediate execution mode: when the lesson sequence is completed, a scheduled interruption occurs at the end of one of the sections, an error is trapped, or the student terminates the lesson prematurely (see item h below).

(g) There should be a central record of student progress on each lesson, and if desired a list of cutoff dates beyond which a student would not receive credit. This record would provide information for grading (if desired) and would include control values for each student, to provide restarting a previously uncompleted lesson at the proper section.

(h) A student should be able to terminate a lesson at any point where a response is called for, by typing STOP. Whenever there is an action terminating a lesson, a student should receive an intelligible message concerning the computer's updating of records. The CAI system should be inoperable if a student's workspace condition is not proper (there is a suspended function, the user number is not valid for operating the CAI system, etc.), and an appropriate message should describe the difficulty. A student should be permitted to do a lesson even if the cutoff date is past, but not for credit, and should also be permitted to do the lesson repeatedly after credit has been received, with the option of selecting particular section numbers (as might be convenient for review).

Example of Operation

The following print-out is a contrived lesson designed to illustrate some features of the system rather than for educational content. It consists of only one numbered section. It is assumed that a library workspace which contains the function *LESSON* has been loaded. Simulated student responses, which follow three dots, are correct for all questions.

```
      LESSON 6
1)  IF YOU HAVE EXECUTED THE STATEMENTS
      D←5
      E←8
AND THEN EXECUTE
      W←D+E
GIVE THE NUMERIC VALUE OF THE ARGUMENT ON
THE RIGHT OF +...8
*CORRECT*
WHAT DOES THE STATEMENT ABOVE DO TO THE
VARIABLE W?...ASSIGNS IT A VALUE
*CORRECT*
REWRITE THE STATEMENT
      2+D×E
SO THAT 2 PLUS THE VALUE OF D WILL
MULTIPLY THE VALUE OF E...(2 + D) × E
*CORRECT*
FOR A STATEMENT WITH MORE THAN ONE
FUNCTION AND NO PARENTHESES, THE EFFECTIVE
ARGUMENT ON THE RIGHT OF ANY ONE OF THE
FUNCTIONS IS (COMPLETE THE SENTENCE)
...RESULT OF EVERYTHING TO ITS RIGHT
*CORRECT*

CONGRATULATIONS:  THE FACT THAT YOU HAVE
      COMPLETED LESSON 6
HAS BEEN RECORDED.  YOU DO NOT NEED TO DO
      IT AGAIN.
```

The answers shown are not the only ones acceptable. For example, in the third question *E*×2+*D* would also be appraised as "correct" (any correct possibility is recognized). The means for dealing with alternative correct answers are discussed later. Also, the third question demonstrates a case where extra spaces are ignored; (2+*D*)×*E* is acceptable.

Some of the answer-checking possibilities are illustrated by the printout shown below, in which all questions are answered incorrectly. Observe the remedial statement and additional question following the first incorrect answer, material that does not appear in the first printout.

```
      LESSON 6
1)  IF YOU HAVE EXECUTED THE STATEMENTS
      D←5
      E←8
AND THEN EXECUTE
      W←D+E
GIVE THE NUMERIC VALUE OF THE ARGUMENT ON
THE RIGHT OF +...E
WRONG: THE CORRECT ANSWER IS 8
VARIABLE E IS ON THE RIGHT OF + AND
HAS BEEN ASSIGNED THE VALUE 8.
GIVE THE NUMERIC VALUE OF THE ARGUMENT ON
THE LEFT OF +...
WRONG: THE CORRECT ANSWER IS 5
WHAT DOES THE STATEMENT ABOVE DO TO THE
VARIABLE W?...NAMES IT
WRONG, TRY AGAIN...GIVES IT A NUMBER
GIVE UP: THE KEYWORD IS SPECIFIES
REWRITE THE STATEMENT
      2+D×E
SO THAT 2 PLUS THE VALUE OF D WILL
MULTIPLY THE VALUE OF E...2+(D×E)
WRONG, TRY AGAIN...(2+D×E)
GIVE UP: THE CORRECT ANSWER IS (2+D)×E
FOR A STATEMENT WITH MORE THAN ONE
FUNCTION AND NO PARENTHESES, THE EFFECTIVE
ARGUMENT ON THE RIGHT OF ANY ONE OF THE
FUNCTIONS IS (COMPLETE THE SENTENCE)
...THE PART TO THE RIGHT
WRONG: THE KEYWORDS ARE: VALUE RIGHT
```

Attention is called to the provision for second tries in some instances, and the use of keywords in two cases. Where it is appropriate, alternative keywords are supplied in the standard answers. All words in the student's response except keywords are ignored. For example, in the question concerning what *W←D+E* does to *W*, the keywords stored as the standard are: *SPECIFIES*, *SPECIFIED*, *ASSIGNS*, and *ASSIGNED*. Thus, some examples of answers that will be accepted as "correct" are

```
SPECIFIES IT
IT IS SPECIFIED
ASSIGNS A VALUE TO W
W IS ASSIGNED A VALUE
```

Details of how alternative keywords are arranged in array structures, and how such structures are employed in checking student answers are given later.

The APL Functions for Presenting a Lesson

A student must load a workspace containing the function *LESSON*, from an appropriate library. Other functions are created automatically from file storage, and remain temporarily in the student's active workspace only while the system is running. The main features of the functions are described in the following:

LESSON

This is a monadic function, taking the lesson number as its argument. It performs the following tasks:

(a) Set traps for exiting the function if there is an error or interruption.

(b) Tie files of student records, subfunctions, and variables (4 files). Read files and fix functions.

(c) Make several checks of operating conditions, such as: validity of student's user number and whether there are suspended functions in the active workspace. If the conditions are not proper, display an appropriate message, terminate the operation, and reload the workspace.

(d) Check whether the deadline date for doing the lesson is past, and check a control variable to determine whether the student has previously done the lesson. If the result of either check is affirmative, display an explanatory message (including "not for credit," if the deadline is past) and give the student an option (by answering YES or NO to a query) as to whether to continue. If the answer is NO, terminate the operation; if it is YES, ask the student for the desired starting section number. Set a control variable for the section number, at the number designated by the student if the lesson is late or a repeat, otherwise at 1.

(e) Execute the subfunction *LESRUN*.

Each lesson is represented in files by two packages. One package consists of a set of what we shall call "section functions" (having names Δ*SF*1, Δ*SF*2, etc.), and the other package consists of a set of "section variables" (named Δ*MS*1, Δ*MS*2, etc.). One section function and its corresponding section variable contain all the information pertinent to the section having the same number as the suffix on the names. Control information is shared by a section function and its corresponding section variable, and the text material is in the variable. Examples for the material illustrated previously are as follows:

```
    ∇ ΔSF1
[1]   MA1←'8'
[2]   RA1←'5'
[3]   MA2←'SPECIFIES⍮ASSIGNS⍮ASSIGNED⍮'
[4]   MA2←FXA MA2,'SPECIFIED'
[5]   MA3←'(2+D)×E⍮(D+2)×E⍮E×2+D⍮E×D+2⍮'
[6]   MA3←FXA MA3,'E×(2+D)⍮E×(D+2)'
[7]   MA4←FXA'VALUE⍮RIGHT⍮⍮RESULT⍮RIGHT'
[8]   1 0 CTRL 7 1 1 2 5 2 3 1
    ∇
```

```
      ΔMS1
    IF YOU HAVE EXECUTED THE STATEMENTS
      D←5
      E←8
AND THEN EXECUTE
      W←D+E
GIVE THE NUMERIC VALUE OF THE ARGUMENT ON
THE RIGHT OF +...
⍟VARIABLE E IS ON THE RIGHT OF + AND
HAS BEEN ASSIGNED THE VALUE 8.
GIVE THE NUMERIC VALUE OF THE ARGUMENT ON
THE LEFT OF +...
∘WHAT DOES THE STATEMENT ABOVE DO TO THE
VARIABLE W?...
∘REWRITE THE STATEMENT
      2+D×E
SO THAT 2 PLUS THE VALUE OF D WILL
MULTIPLY THE VALUE OF E...
∘FOR A STATEMENT WITH MORE THAN ONE
FUNCTION AND NO PARENTHESES, THE EFFECTIVE
ARGUMENT ON THE RIGHT OF ANY ONE OF THE
FUNCTIONS IS (COMPLETE THE SENTENCE)
...
```

A section function such as ΔSF1 carries specifications of standard answers, in terms of "answer variables" *MA*1, *MA*2, etc., and *RA*1. Those beginning with *M* are answers to main (primary) questions, those beginning with *R* are answers to remedial questions. The function *FXA* appearing within ΔSF1 converts its vector argument to an array of alternative correct answers or keywords in accordance with the locations of the separator symbol ⍪ (I-beam symbol). *FXA* is effectively equivalent to the reshape function, but does not require the inclusion of filler spaces in its argument. If one does not mind counting characters, reshape can be used instead. Further details about *FXA* and how answer variables are structured are given later. Those produced by *FXA* in ΔSF1 are

```
      MA2
SPECIFIES
ASSIGNS
ASSIGNED
SPECIFIED

      MA3
(2+D)×E
(D+2)×E
E×2+D
E×D+2
E×(2+D)
E×(D+2)

      MA4
VALUE
RIGHT

RESULT
RIGHT
```

A section variable such as ΔMS1 contains one or more items of primary instructional text and questions, and optional items of remedial material. Primary text material is identified by a control symbol ∘ at the beginning of the first line of each item except the first. A remedial item is always identified by the control symbol ⍟ at the beginning of the first line. (In the present system, two ⍟ symbols cannot appear in sequence without an intervening ∘ symbol.) These control symbols are suppressed when the text is presented to a student.

The section and answer variables are global to the control function *CTRL*, which is executed at the end of a section function. The left-hand argument of *CTRL* has two elements: the first is the section number, the second is 0 if the corresponding section is the last one in the lesson, or 999 if the lesson continues but there is a planned return to the immediate execution mode (as when the student is instructed to originate some APL operations), or otherwise 1.

The right-hand argument of *CTRL* consists of one pair of nonnegative integers for each main question. The first integer of a pair is a specification of the mode (described later) to be used by the computer in checking a student's answer against the standard answer, and the second integer indicates the number of tries a student should be allowed on the question. The specification for a remedial question is not included explicitly; by implication it is the same as for the main question to which it relates.

Functions *LESRUN*, *FXA*, *CTRL*, and *CHECK* (and some other subfunctions) are created from the files when *LESSON* is executed. The main tasks performed by these are as described below.

LESRUN

(a) Read the value of a control variable *K* from a file; the number of the last section completed, and other control information.

(b) Execute
```
      ⍎'ΔSF',⍕K←K+1
```
until the second element of the left-hand argument of *CTRL* in the current section function is 0 or 999, or the student signals STOP.

(c) Update the files with new information about the student's progress, including the time of day and date.

(d) Print an appropriate message to the student regarding conditions of the termination.

FXA

(a) If the argument contains no ⍪ symbol (except possibly at the end), return the value of the argument unchanged, but with any trailing ⍪ removed.

(b) If the argument contains one or more I symbols (not counting any at the end), expand the argument with any required filler spaces, remove the I symbols, and form a matrix in which each I marks the end of a line. *FXA* interprets the argument as ending with I, but the explicit inclusion of I at the end is optional.

(c) If the argument contains one or more II pairs (not counting any at the end), expand the argument with any required filler spaces, remove the I and II symbols, and form an array of rank 3 in which each I marks the end of a line, and each II pair marks the end of a page (plane). *FXA* interprets the argument as ending with II, but the explicit inclusion of II at the end is optional.

CTRL

(a) Analyze the corresponding Δ*MS* variable (identified by the first element of the argument on the left), separating it into parts on the basis of locations of the control symbols ∘ and ⊗ at the beginnings of lines.

(b) Print text material pertinent to a question, and the question. Execute the function *CHECK*, which accepts a student's answer and checks it in the mode determined by the appropriate pair of elements from the right-hand argument. If the student's answer is wrong, print remedial text and question (if any); and if there is a remedial question, execute *CHECK* in the same mode. Continue in this manner until all parts of the text in the section variable have been exhausted, or the student enters STOP as an input.

(c) Set a control variable for *LESRUN* which indicates whether execution of *LESRUN* should continue to the next section (*LESRUN* is not continued if the student replies with STOP, or the second element of the left-hand argument is 0 or 999).

(d) Return execution to *LESRUN*.

CHECK

Accept character input from a student. If the student's input is STOP, set returned value to ¯1 and exit; otherwise check the student's input against the standard provided by the appropriate *MA* or *RA* variable, in the mode transmitted to it from the right-hand argument of *CTRL*. Print *CORRECT* or, if the answer is wrong, print a message appropriate to the mode of checking. Set the returned value to 0 if the answer is wrong, or 1 if it is correct.

Modes for Checking Answers

The first digit of each successive pair in the right-hand argument of *CTRL* is a control digit that specifies the type of check to be made on a student's answer. Prior to comparing with standard answers, any trailing punctuation mark is removed from the student's response. Then, a check is made against the standard answer in accordance with the mode corresponding to the control digit. These modes are described as follows (with the control digit shown on the left):

0: Extra spaces and punctuation marks are removed from the student's reply. The standard is a vector or matrix of one or more acceptable answers. Complete agreement is required.

1: All punctuation marks are removed from the student's reply, and individual words of that reply are put into separate rows of a matrix. Each row of this matrix is checked against each row of the standard matrix of keywords. The answer is correct if there is a match for any pairing.

2: The student's input is modified as in case 1, but the standard contains two kaywords in preferential order. If there are alternatives, the standard is of rank 3, with alternatives for the first keyword in the first rows of the pages (planes), and alternatives for the second keyword in other rows. The result is correct if the student's input contains any of the alternative first and second keywords in the proper order.

3: The student's input is modified as in case 1, but is checked for N keywords against a standard of rank 2 or 3 (rank 3 when there are alternative keywords) in such a way that if any keyword from each row of the standard is found in the answer, the answer is correct. The order of the keywords is not significant.

4: This is similar to case 0, except that extra spaces and punctuation marks are not removed. This case was designed to deal with exercises in function editing. Accordingly, if the computer prints the correct answer, it begins at the margin.

5: All spaces are removed from the student's input, but punctuation marks are retained. An exact check against a vector or any row of a matrix standard is required. This case is useful in checking APL statements, in which spaces are not relevant.

6: Similar to case 0, but punctuation marks are not removed.

7: Makes a numerical check of the student's answer against the standard.

For those modes involving keywords, if a negative word such as NOT is included in the student's response, this word will be rejected if it is not a keyword, and therefore an incorrect evaluation will be made. Accordingly, a special check is made against a list of words which imply negation, and if one is found that is not a keyword, the answer is rejected with an appropriate message.

In cases where there are alternatives in the standard, the answer presented by the computer as "correct" when a student is unsuccessful on the question is the first one listed in the standard. Therefore, if there is a preferred answer, that one should occur first in the answer variable.

Creating New Text Material

The files for section variables and section functions, for a new lesson, are initialized as packages of an empty vector. New sections are added, one at a time, by the use of two functions: NEWTEXT and INSERT. An example of the creation of the sample section displayed previously is as follows:

```
      NEWTEXT
NEW OR EDIT? NEW
*     ΔA 1
[1]       IF YOU HAVE EXECUTED THE
                                STATEMENTS
[2]         D←5
[3]         E←8
[4]   AND THEN EXECUTE
[5]         W←D+E
[6]   GIVE THE NUMERIC VALUE OF THE
                               ARGUMENT ON
[7]   THE RIGHT OF +...
[8]   ⍟VARIABLE E IS ON THE RIGHT OF + AND
[9]   HAS BEEN ASSIGNED THE VALUE 8.
[10]  GIVE THE NUMERIC VALUE OF THE
                               ARGUMENT ON
[11]  THE LEFT OF +...
[12]  ∘WHAT DOES THE STATEMENT ABOVE DO TO
                                       THE
[13]  VARIABLE W?...
[14]  ∘REWRITE THE STATEMENT
[15]        2+D×E
[16]  SO THAT 2 PLUS THE VALUE OF D WILL
[17]  MULTIPLY THE VALUE OF E...
[18]  ∘FOR A STATEMENT WITH MORE THAN ONE
[19]  FUNCTION AND NO PARENTHESES, THE
                                EFFECTIVE
[20]  ARGUMENT ON THE RIGHT OF ANY ONE OF
                                       THE
[21]  FUNCTIONS IS (COMPLETE THE SENTENCE)
[22]  ...
[23]  ****
[24]  MA1←'8'
[25]  RA1←'5'
[26]  MA2←'SPECIFIES⍪ASSIGNS⍪ASSIGNED⍪'
[27]  MA2←FXA MA2,'SPECIFIED'
[28]  MA3←'(2+D)×E⍪(D+2)×E⍪E×2+D⍪E×D+2⍪'
[29]  MA3←FXA MA3,'E×(2+D)⍪E×(D+2)'
[30]  MA4←FXA'VALUE⍪RIGHT⍪⍪RESULT⍪RIGHT'
[31]  1 0 CTRL 7 1 1 2 5 2 3 1
[32]
*     →→→→
```

This portrayal is distorted because of the column width limitation of this document. The words shown at the ends of unnumbered lines would really be at the ends of the lines above. Also, with longer lines, *MA*2 and *MA*3 could each be specified on one line.

The main feature of *NEWTEXT* is that the *ΔEDITOR* text editor functions are employed.[2] These operate in the immediate execution mode, and are executed within *NEWTEXT* through the subfunction.

```
    ∇ CALCSIM;B;⎕TRAP
[1]   ⎕TRAP←'∇ 0 E ''ERROR, AGAIN''◇→L1'
[2]  L1:⍞←6↑'*'
[3]   →(∧/'→'=4↑B←6↓⍞)/0
[4]   ⍎B
[5]   →L1
    ∇
```

which simulates the immediate execution mode, except that * appears at the margin as a prompt symbol, and normal error reports and suspensions are suppressed in favor of the general message "ERROR, AGAIN" followed by retyping of the prompt symbol. Typing →→→→ causes exiting from *CALCSIM*.

The question NEW OR EDIT? refers to whether a new section is to be written, or editing of a new section <u>not yet in the files</u> is desired. The *ΔA* 1 opposite the prompt symbol is an execution of the *ΔEDITOR* function for adding lines, starting with line 1. The numbers in brackets are supplied by *ΔA* but do not appear in the resulting matrix. The return without input at [32] causes exiting from *ΔA*, and →→→→ causes exiting from *CALCSIM*, allowing *NEWTEXT* to run to completion. In this mode (NEW is the reply to the initial question) *NEWTEXT* initializes an empty matrix which is expanded as input lines are typed, and upon completion of the text entries *NEWTEXT* causes this matrix to be stored in a temporary file.

In the mode of *NEWTEXT* when EDIT is the reply to the initial question, the matrix in the temporary file is brought back into the workspace so as to be available for modification through the various editing functions of *ΔEDITOR*. For example, line [9] can be replaced by a different version by executing

```
      NEWTEXT
NEW OR EDIT? EDIT
*     ΔR 9
[9]   HAS THE VALUE 8.
*     →→→→
```

The details of *ΔEDITOR* are too extensive to be described here; suffice it to say that it is very flexible, permitting selective editing of lines, insertions, deletions, and display.

When the text is ready, it is inserted in its proper position in the lesson sequence by executing the dyadic

function *INSERT*. The arguments on the left and right are, respectively, the lesson number and the position in the sequence of sections where the new section is to be inserted. As a precaution, there is a check to ensure that the right-hand argument of *INSERT* agrees with the first element of the left-hand argument of *CTRL* which appears on the last line of the text. *INSERT* can be used to imbed a new section within an existing sequence, and causes the proper renumbering of those existing sections which follow it.

INSERT reads the text created by *NEWTEXT* from the temporary file, produces the variable Δ*MS*1 from the material above the separator ****, and creates the function Δ*SF*1 from the material below the separator. These are then incorporated in the appropriate packages in the two files mentioned earlier. A test is included to ascertain whether arguments of the *FXA* functions appearing in the section functions (such as Δ*SF*1) are properly formed.

In order to specify the answer variables appearing below the separator, the person writing the text must know how to choose a shape for the array appropriate to the desired answer-checking mode, and be able to position I markers in the arguments of the *FXA* functions so as to yield that shape. Possibly, catenation will be needed in forming these arguments. It is noted that all answers are in character form, even if the check is to be numeric. This is for the convenience of the writer, providing the uniformity of having all answers in character form. The use of *FXA* in those cases where an answer variable has a vector or scalar value is optional. If there are no I symbols in the argument of *FXA*, it returns its argument unchanged.

Revising Old Text Material

A section can be deleted from a lesson by executing the function *REMOVE*. It takes the lesson number as the argument on its left, and the section number on its right. Remaining sections are automatically renumbered.

A function *REVISE* is available for modifying any section which has previously been inserted in the files. It takes the same arguments as described above for *REMOVE*. When *REVISE* is executed, it recombines the material of the corresponding Δ*MS* and Δ*SF* objects, into a single matrix, and executes *CALCSIM*, allowing the functions of Δ*EDITOR* to be used for displaying and making revisions. When →→→→ is entered, to terminate *CALCSIM*, the execution of *REVISE* goes to completion, remaking the Δ*MS* and Δ*SF* objects and reconstituting the packages in the files. Arguments of the *FXA* functions are checked for validity in the same manner as with *INSERT*.

It would be beyond the scope of this paper to discuss the pros and cons of computer aided instruction, but it is suggested that a system such as this can be usefully applied in certain clearly defined portions of many subjects (not necessarily computer programming) which require a large amount of student interaction.

The system has been used for two years at Syracuse University (with a total of about 600 students) in connection with the APL course mentioned in the introduction. There are five lessons, each taking about one-half hour of a student's time, on variable names and values, workspace management, the execution rule, function editing, and structure of arrays (done in that order). Acceptance of the system by the students is very high. Since the introduction of these lessons, it has been possible to drop a "workshop," previously required as an aid in getting students started, with a concomitant acceleration of student progress. In this instance students are not graded on answers to the questions (only on the amount done) because the intent is to encourage them to do the lessons very early in the course; some students would have anxiety and procrastinate if they were graded on answers. However, the system could easily be changed so as to record right and wrong answers on a single pass through a lesson, or to permit multiple passes and record ultimately attained correct answers.

The modes for checking answers described in the text comprise a modestly sophisticated system that is adequate for the purpose for which the system was designed, where there is no grading of correctness of answers. Undoubtedly there are occasions when substantially correct student responses are incorrectly appraised, but there have been absolutely no complaints from students. Apparently the number of incorrect evaluations is tolerable. However, the design permits the inclusion of additional checking modes, merely by modifying the function *CHECK*. The present policy of making exact comparisons of keywords is not adequate if students are graded on their responses. Many students are poor spellers or poor typists, but usually one would not want to penalize them for those reasons. This is a fundamental and difficult problem for which some modest improvement can be envisioned.

Concerning experience with the design and realization of the system itself, workable APL programs for a rudimentary realization of the original design were realized is only a few days of programming. Subsequent revisions have been incorporated,

but with practically no necessity for rewriting anything but the part being changed.

References

(1) APL as a Coursewriting Language, R.W.W. Taylor; APL Quote-Quad, June, 1972, pp. 3-12.

(2) ΔEditor-APL Function and Data Maintenance System, R.G. and J.W. Burgeson; APL 76 Conference Proceedings, pp. 166-177.