ED 125 599 IR 003 686

AUTHOR	Danielson, R. L.; And Others
TITLE	Final Report on the Automated Computer Science Education System.
INSTITUTION	Illinois Univ., Urbana. Dept. of Computer Science.
PUB DATE	Jun 76
GRANT	NSF-EC-41511; NSF-EPP-74-21590
NOTE	51p.; Period of project: January 1, 1974 to June 30, 1976
EDRS PRICE	MF-$0.83 HC-$3.50 Plus Postage.
DESCRIPTORS	Annual Reports; *Computer Assisted Instruction; *Computer Science Education; Higher Education; Independent Study
IDENTIFIERS	*PLATO; Programmed Logic for Automatic Teaching Operations; *University of Illinois Urbana

ABSTRACT

At the University of Illinois at Urbana, a computer based curriculum called Automated Computer Science Education System (ACSES) has been developed to supplement instruction in introductory computer science courses or to assist individuals interested in acquiring a foundation in computer science through independent study. The system, which uses PLATO terminals, is presently in routine use in several courses at the University of Illinois, and it has been used at Wright Community College in Chicago. Recent changes in programing and technical innovations have increased its instructional effectiveness. The first section of this report describes the goals and design of ACSES. Later sections provide yearly reviews of progress made for the duration of a grant from the National Science Foundation. (EMH)

NSF-TIE-EPP-74-21590

FINAL REPORT

ON THE

AUTOMATED COMPUTER SCIENCE EDUCATION SYSTEM

Grant Nos.: NSF-EC-1511 and NSF-EPP-7--21590

Period of project: January 1, 197- to June 30, 1976

Submitted by: P. L. Danielson
H. G. Friedman, Jr.
W. J. Hansen
P. G. Montanelli, Jr.
J. Nievergelt
T. R. Wilcox

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois 61801

TABLE OF CONTENTS

1. Introduction

This first section provides a brief description of the design and goals of the Automated Computer Science Education System (ACSES), and a summary of its current status. The next three sections provide yearly reviews of progress made toward this current status during the duration of the grant. Section 5 is a bibliography of papers and reports produced by members of the ACSES project.

A slightly more detailed summary of the principle components of the system is contained in [Nievergelt, J., et al, 1976]. Complete details of various components may be found in one or more of the publications listed in section 5.

1.1. Review of the project and its goals

Over the past three years, the Department of Computer Science has been heavily engaged in a project to develop on PLATO an instructional system capable of assuming a large part of the teaching load in all its introductory programming courses. Beginning January 1, 1974, this project has been partially supported by NSF under grants EC-41511 and EPP-74-21590.

The major goals of the project were:

-- the system should be useful as a supplement to classroom instruction in any introductory computer science course--no matter what programming language is taught, what examples of computer applications are discussed, and what the instructor's philosophy is about how programming should be taught.

-- the system should be able to provide at least half of the instruction in any one of our own introductory CS courses (currently, C.S. 101, 102, 103, 105, 106, 107, 121, and C.S. 400).

-- the system should be usable without an instructor by anyone having access to a PLATO terminal who wants to learn the beginnings of computer science on his own.

One can summarize these points by saying that the instructional system should be able to operate in a partially automated mode (with the degree of automation determined by the instructor) as well as in a fully automated mode.

These goals have caused us to organize the instructional system as a collection of relatively independent and self-contained components. The objective has not been to take one of our existing CS courses and put it on PLATO as it is. Rather, we wanted to turn a PLATO terminal into a rich environment, analogous to a conventional library and laboratory, where one has at one's fingertips many useful things for learning about computer science, and for practicing immediately what one has learned. It is the student and his instructor who decide which of these things they want to use.

The principle components of ACSES which allow this flexibility are: The lesson library, with a classification scheme in which any self-contained lesson about introductory computer science will fit; the guide, an information and advising system which gives the user access to this library and allows an administrator to keep track of all his lessons and their interrelationships; and the programming system, on which an additional programming language can be implemented in a few man-months (by somebody familiar with the system), and in which all user-dialog (program entry and editing, error-analysis, and execution modes) is carried out in a uniform way, independent of the particular programming language involved.

Last but not least, the project had one more important goal:

-- to serve as a stimulating environment for computer science research in a variety of areas: compilers, information systems, artificial intelligence.

5

This last point may deserve some explanation, since the misconception is widespread that lesson writing is a routine activity. It can be, if one creates poor lessons. It can also be a task as challenging as one wishes to make it, if one views a lesson as an interactive program that has a certain domain of knowledge, and is able to communicate with students about the knowledge it has.

1.2. Summary of current status

Our experience with ACSES, and the status of the major components of the system, may be summarized as follows:

1. The system is in routine use for several of our introductory programming courses (C.S. 101, 102, 103, 105, and 400). In C.S. 103 and 105, ACSES is used to replace one lecture each week with each student spending about one hour weekly at a PLATO terminal; C.S. 101 and 102 have so far used ACSES as supplementary material only; C.S. 400 is taught primarily via ACSES. All told, nearly 1,300 computer science students per semester are making contact with PLATO.

Controlled experiments have indicated that students in PLATO sections learned as much as those in non-PLATO sections, and would strongly recommend PLATO sections to their friends, but dropped at a somewhat higher rate.

As an indication of the acceptance of PLATO within our department, in Fall 1976, C.S. 101 and C.S. 105 will each have one less professor assigned than usual, with the remaining instructional load taken up by ACSES.

(2) After considerable effort, we have been successful in getting another institution to make regular use of ACSES. During the academic year 1975-76, over 350 students in five sections of three courses at Wright Community College in Chicago have received part of their instruction by way of our system.

(3) The lesson library has grown to about 135 lessons, grouped into about 20 areas, and continues to expand slowly. In addition, we are making systematic efforts to improve the quality of the lessons based on our experience with their use in instruction.

(4) The computer assisted programming system (CAPS) is operational and runs PL I, FORTRAN, COBOL, PASCAL, BASIC, and LISP programs. CAPS provides automatic analysis of errors at both compile-time and run-time, interacting with the student to help him find the cause of his error. Unfortunately, the usability of this system has been severely hampered by the processing limitations imposed by PLATO.* We have thus devoted a great deal of study and work to developing techniques to improve the efficiency of the system.

(5) The information and advising system, csguide, is complete. The guide is designed to make ACSES usable by people who want to study on their own, by advising them on their choice of lessons based on their goals and past performance. It communicates primarily by means of simple English phrases as input, and pictures for output, where the relationships among lessons of relevance to the user are displayed in graphic form.

In another use of ACSES material outside our group, the guide has been adopted by the PLATO foreign language group to administer their collection of lessons.

(6) An interactive exam system, designed for automatic generation and administration of exams, has been completed, although there is still need of additional problem generators and graders. We have conducted several experiments involving the exam system, and are conducting another in Summer 1976. As a part of the exam system, a quiz system designed to assess and control the student's progress through various lessons has been implemented. Many lessons have had such quizzes written, and we are continuing to write more.

(7) Experimentation on the design of control structures for interactive programming languages, particularly in CAI, has been completed. This work has led to the development of several principles (uniformity, separability, and locality) which are proposed as a partial basis for the design of programming language features.

(8) Two lessons concerned with judging student programs have been essentially completed, and work on determining techniques for detecting various anomalies in student programs by global flow analyses has been significantly advanced.

2. Activities: January, 1974 - June, 1974

2.1. The library of lessons

During this period the library has grown from 50 to about 90 lessons which are in some sense "operational", and, more importantly, many of the lessons have been improved. Much of this effort (by H. G. Friedman, R. G. Montanelli, D. Eland and S. Leach) has been directed toward polishing elements of the FORTRAN sequence and improving csrouter in preparation for the PLATO experiment in CS 103 this fall.

Among the improvements to the lessons has been a certain amount of standardization. For example, the function of the keys that allow a student to control his path through a lesson has been standardized among all CS lessons, so that a student who has gone through a few lessons has become familiar with most of the control options in all lessons and can proceed through subsequent lessons more easily. These conventions are described in lesson csauthors (F. Izquierdo). Standard pieces of code, character sets, and micro tables have been collected in lesson cslibrary (H. G. Friedman).

Also, several communication lessons have been completed. Lesson csnotes serves a "bulletin board" function between authors. Lesson cscomments (D. Eland, E. Reingold) serves to record remarks made by students taking instructional lessons, for feedback to the authors of the lessons. Lesson cstalk (H. G. Friedman) allows real-time communication between an instructor and each of several students; this allows human assistance to a student by an instructor who might be located at a different PLATO site.

2.2. Computer assisted programming system (CAPS)

A table-driven program entry and editing and syntax analysis program has been completed (T. R. Wilcox, A. Davis, M. Tindall). Tables for PL/1 are complete. Tables for FORTRAN, BASIC, COBOL, SNOBOL, APL, and MIX have been started. Program entry and editing are working for FORTRAN, COBOL, and BASIC.

The run time system for each of these languages is different. The PL/1 run time system works for a subset of PL/1 suitable for the first half of one semester's instruction, and is being used this semester in one introductory course, CS 106. Run time systems for FORTRAN and BASIC are in progress, and COBOL has been completed.

M. Tindall has implemented a prototype version of an automatic syntax error analysis scheme; this prototype system is installed in a developmental version of the general compiler system and works moderately well on the PL/1 language and tables. He is currently attempting to achieve this same working level with the FORTRAN language and tables (and, perhaps, the COBOL language). When this is accomplished, he will revise and refine the prototype model to obtain a stable, operational version of the compiler with the automatic error analysis system included.

A. Davis has expanded his execution supervisor for PL/1 to include character variables and their associated operators and built-in functions: concatenation, INDEX, SUBSTR, LENGTH, and VERIFY. The necessary changes were also made to the PL/1 syntax tables. Because of space limitations imposed by PLATO, it seems unlikely there will be any immediate expansion of the subset of PL/1 executed. Presently, the statement types accepted include PROCEDURE, DECLARE, assignment, IF THEN ELSE, GOTO, DO (all forms), GET LIST, PUT LIST, and END.

About 80% of the execution-time error analysis system has been coded and debugged. Despite delays caused by the previously mentioned space limitations, completion of the run-time analysis package is expected before the end of the calender year.

B. Segal is conducting a study to explore the effect of system interruptions on user performance at a terminal. Our compilers interrupt the student who is typing a program at the very moment he makes an error that can be detected by the compiler, and we would like to know whether or not this is a more effective way than the conventional approach, where all errors are signaled when the program has been entered completely.

A pilot study has been completed and results show some statistically significant differences between the two error interruption methods; a full-scale experiment will be carried out in October.

2.3. Information and advising system

A version of the GUIDE conversational information system, designed to advise students in their choice of lessons based on their goals and past performance, is currently operational. A student can type in a request for information (an English sentence), and the system will respond by displaying the desired information (or an explanation will be given as to why the request could not be handled).

The first part of the GUIDE, the translator, has been completed as part of a Ph.D. thesis by J. Pradels. A request in English is translated into an internal request language by means of a finite state automaton. This internal representation is translated back into English, so that the user can judge for himself whether his request has been properly understood.

D. Eland has implemented a request processor which analyzes the intermediate representation of the original request, searches the proper data bases for the desired information, and generates a display which presents an appropriate response to the student. In addition, he has completed detailed design of the entire data base for the system. A data base editor has been implemented which facilitates the entry of information, maintains internal consistency, and manages the allocation of storage. Information on each lesson in the library and several course outlines have been entered in the data base, and routines to manage collection and display of student record data have been written.

Current work on the system includes monitoring its performance under student use, development of a thorough description of each lesson in the library (with particular emphasis on the structure of relationships between lessons), and research into the content and structure of the concept space describing the discipline of computer science.

2.4. Exam system

D. Eland has written a program called examadmin to handle all security and data collection aspects of exams given on PLATO. He has also written a prototype PL/1 exam. Two efforts have been started toward automatic problem generators. F. Izquierdo has nearly completed a table-driven generator of single statements, of specified type and complexity, in any of the programming languages implemented in the CAPS interactive programming system. This program will be used together with a question generator and related grader, to produce a variety of test questions using single statements.

B. Barta has begun work on an interactive system for automatic examination of programming skills using the CAPS system for checking

12

and grading programs prepared by students. The student taking the exam types in a program as a solution and keeps interacting with the system until either the system accepts the solution as correct or the student decides to quit. The sequence of interactions and the grading criteria are partly specified by the problem author and partly built into the system. The instructor prescribing the examination will have some control on the scoring criteria.

2.9. Automatic judging of student programs

Work is underway on two lessons which ask the student to write fairly sophisticated programs and attempt to judge these programs interactively.

One of these (R. Danielson) asks the student to write a PL/1 program for symbolic differentiation, and is operational in a prototype form. Current efforts are concerned with refining the prototype's reaction to student inputs. Two approaches are being taken: (1) protocol analysis of sessions on the symbolic differentiation problem between introductory-level students and human tutors, as an aid to determining valid solutions and aligning the natural language understanding routines with statements used by beginning programming students, and (2) closely supervised student use of the machine tutor, to improve the operational aspects of the student-tutor interface.

The other (P. Mateti) is intended to be able to judge arbitrary sorting programs written in a specialized language. An editor and interpreter have been written for this special programming language, and for an assertion language in which assertions can be made about the state of the array to be sorted. These routines are being reviewed in an attempt to improve their efficiency, and design efforts have begun on a theorem prover which will use the assertions to prove or disprove the correctness of the sorting program.

2.6. The KAIL project

The objective of the KAIL project is to propose an alternative to the TUTOR language in the PLATO environment. In designing KAIL, emphasis has been placed on providing a good structure for the language since this is the area in which TUTOR is weakest. Thus, KAIL is an ALGOL-like superset of TUTOR with procedures, block structure, and scope rules for variables, modifications, and user defined interrupts.

A subset of KAIL was chosen for implementation and a compiler has been designed, implemented, and debugged (D. Embley, W. Hansen). The output of the compiler is TUTOR code which is condensed and run on the PLATO system. Experiments are being designed to determine if these new features are easier to understand and use than traditional features, as represented in TUTOR.

2.7. Use of ACSES in instruction

The system has been used in several courses. The most extensive use involved about sixty students (one-half the class) who learned PL/1 on PLATO in CS 121 in the fall of 1973. This two month trial was used to test procedures for a later controlled experiment and to evaluate the existing PL/1 lessons. Tentative conclusions from the experiment were that students reacted favorably to PLATO and learned as much as students in the regular sections, even though the lessons were still in need of improvement. This latter finding is only a slight indication that the existing lessons were on a par with the teaching assistants in teaching PL/1, as students were not randomly assigned to groups, and other experimental controls were lacking.

In the spring of 1974, the instructional system was used in an auxiliary mode in two introductory courses, CS 103 and CS 106, and also in CS 211.

15

3. Activities: July, 1974 - June, 1975

3.1. The library of lessons

We currently have over 100 lessons, grouped into about 20 areas. The majority of them were created by students as term projects in a course on computer-assisted instruction, or as other academic activities, such as theses. Considering this origin, it is understandable that many of these lessons are of poor quality. Considering their number, it is also understandable that we do not have the manpower to bring all of them up to a sufficient level of quality.

The need for repeated revision based on experience with actual instruction, and the significant amount of effort required for each revision, are the causes for the fact that only a fraction of our lessons are anywhere close to their "final form". We have many lessons that have never been used in instruction, and are unlikely to be used in our own courses in the near future, since the library has grown to a size beyond our needs.

Until further uses of ACSES appear, we have concentrated most of our resources for lesson improvement on the following core of about 50 lessons:

1) FORTRAN sequence (used in CS 101, CS 103, CS 105, CS 400).
2) PL/1 sequence (used in CS 106, CS 107, CS 121, CS 300).
3) Lessons on computer applications suitable for these courses.
4) Language-independent lessons on programming principles.

Also, we plan to introduce into ACSES some tighter means of controlling the student's progress through his course, and giving him certain options only after he has completed important assignments.

Because of our goal to create an instructional system that could be accepted in any school, ACSES has been designed according to the philosophy appropriate for an information system: if someone is sufficiently motivated to do so, he should be able to extract from the system the relevant information he wants; the system should offer a large number of choices, and it should never impose itself by forcing the user to do anything he may not wish to do. This philosophy is appropriate for some uses of an instructional system (e.g., adult continuing education), but a more structured system is likely to be more effective for teaching large introductory courses that are required, where a large number of students are not self-motivated.

H. G. Friedman manages the library of lessons, with the assistance of S. Leach. The FORTRAN sequence, which is our best tested set of lessons, is supervised by R. G. Montanelli, Jr.

3.2. Computer assisted programming system (CAPS)

The table-driven programming system is operational and runs for subsets of PL/1, FORTRAN, and COBOL suitable for a first programming course. The FORTRAN processor, implemented by M. Milner, was used last semester by 600 students in CS 105 for about one hour per week. The COBOL processor was implemented as part of an M.S. thesis by R. Barnett. As part of their Ph.D. research, A. Davis and M. Tindall have completed an error-analysis system (for run-time and syntax errors, respectively), which carries out a dialog to help the student find the cause of the error and correct it.

Pl/1, FORTRAN and COBOL constitute the core of our programming system that must be maintained in reliable form. Other efforts are going on which are not essential for the use of the programming system in our own courses: implementations of Pascal, BASIC, and LISP, and a table-driven interpreter. These may or may not lead to useful products in the near future.

The efficiency of the programming system is of continuing concern to us. Prior to the expansion of PLATO's Extended Core Storage from 1 million words, the memory requirement of the programming system made its use difficult, but since February this problem has disappeared. Now the CPU-time requirements of the programming system make its use unpleasant when there are 400 active terminals. We have improved the efficiency of the programming system in a variety of ways, for example by adding a mode where syntax is performed line-by-line instead of character-by-character. This version of FORTRAN is being used this semester by 450 CS 105 students. CERL has cooperated in this effort by introducing a new TUTOR feature, "vertical segment", which makes table-look-up operations much faster. Despite these efforts, the programming system often forces the student to go slower than he could. E. White is currently engaged in a careful analysis of the FORTRAN processor in an attempt to isolate and then improve the time-consuming portions of the system.

3.3. Information and advising system

The csguide information system has been completed by D. Eland as part of his Ph.D. thesis. csguide communicates primarily by means of simple English phrases as input, and pictures for output, where the relationships among lessons of relevance to the student are displayed in graphic form.

csguide was designed to make ACSES usable by somebody who wants to study on his own. Where students are formally enrolled in our CS courses, the instructor carries out many of the functions for which csguide was designed, and hence csguide has not been used much so far. It will only serve its purpose if ACSES attracts an audience at remote sites.

The csguide data base is also used in the routing and control of students using the computer science lessons on PLATO. A few additions to this data base have therefore been made by H. G. Friedman to enable more flexible routing of students. Also, the information maintained in the routing process about each student's progress has been made available in a lesson, csrecords, in a form in which it can be reviewed by an instructor.

The guide's ability to respond to student requests and quickly and concisely present information via graphic displays has resulted in the core routines being adopted by the PLATO foreign language group to administer their collection of lessons.

3.4. Exam system

As part of the emphasis to develop those aspects of ACSES that are of direct use in our own courses, various programs that have to do with exams (an interactive program grader written by B. Barta, a problem generator by F. Izquierdo, an exam administrator by D. Eland) have been integrated into an exam system by L. Whitlock. The system is organized around a central monitor responsible for record keeping, and uses a number of "problem generator/graders", each responsible for generating a problem and grading the answer.

Two exams for CS 101 were administered using the exam system during summer, 1975. R. Doring, L. Whitlock, and W. Hansen have conducted an experiment this fall to compare the time students spend taking an exam on PLATO with that spent on a pencil and paper exam. These experiences have been used to improve the human interface of the exam system.

There are currently several people working on additional problem generators for the system, some of which will accept a "difficulty" parameter and generate a problem of that relative difficulty. Because of the wide variety of problems that will be available, the system can be used for exams, for student review, and for experiments. We intend to explore the acceptability of exams where the difficulty of problems is varied during the exam in reaction to previous responses.

3.5. <u>Automatic judging of student programs</u>

This past year we have continued to work on two lessons for judging student programs, and have just begun another effort in that direction.

The recently completed lesson by R. Danielson tries to follow the student's thought processes as he develops a PL/1 program for symbolic differentiation of expressions by the technique of stepwise refinement of the problem. An AND-OR graph is used as a model of the refinement process, and is traversed by the lesson in the course of monitoring program development.

As part of his thesis research, P. Mateti is completing a program which will be capable of proving the correctness (or incorrectness, with counterexample) of an arbitrary sorting program written in a language with specially designed sorting primitives (e.g., <u>interchange</u>)

20

and including, at various points in the program, appropriate assertions about the state of the array being sorted. A program entry and monitoring module has been completed which provides a dynamic display of program execution. A special theorem prover, which is highly efficient for the restricted domain of programs being considered, is being implemented to complete this lesson.

A third project in judging student programs has recently been started by W. Gillett. It is concerned with those legal programming constructs which probably indicate a lack of understanding by the student (e.g., B**1/2, which is equivalent to B/2). A study is underway to identify and categorize these "conceptual errors", and a system is being designed which will scan arbitrary student FORTRAN programs for such errors, describe them to the student, and provide advice and hints on how they should be corrected.

3.6. <u>The KAIL project</u>

D. Embley and W. Hansen have developed a high-level author language, KAIL, and written a preprocessor which translates KAIL into TUTOR. Several lessons have been written in the new language. KAIL adds structured control facilities to the set of TUTOR commands by means of the "selector", which combines flow of control with answer judging, and subsumes most of the currently popular control constructs (e.g., <u>if-then-else</u>, <u>case</u>, <u>cond</u>, <u>while</u>).

The results of a recent experiment indicate KAIL's version of selection and iteration is easier to understand than more traditional syntax. These results are summarized in a technical report. Another experiment is in progress to compare KAIL's proposed means of lesson and help sequencing with those of TUTOR.

3.7. Use of ACSES in instruction

During the past academic year ACSES has been systematically introduced into our introductory courses: first in a small class for social science students (CS 103), in order to class-test the FORTRAN lessons; then a large course for commerce students (CS 105). In the first case, a controlled experiment turned out unfavorable to the PLATO group, for reasons we believe we understand, and have corrected. In the second case, results were favorable both with regard to student performance and attitude. We are trying to validate this favorable impression by means of a controlled experiment this fall.

These experimental evaluations have been carrried out by R. G. Montanelli, Jr., some in cooperation with Esther Steinberg of the CERL evaluation group.

They are now described in more detail.

(1). In the Fall Semester 1974, CS 103 (a class of about 60 students) was randomly divided in half for an experimental evaluation of the FORTRAN lesson sequence. The control group received two lectures and one discussion (a meeting in a small group of about 15 students, with a teaching assistant) per week, as the course has always been taught in recent history. The experimental group had one of the lectures replaced by an hour on PLATO, to learn FORTRAN. The two groups were then compared on various measures during the semester.

Results indicated that the experimental group performed worse on all three exams during the semester, although there were no differences on scores on computer programming problems. However, students remained interested in and enthusiastic about PLATO in spite of their deficit in performance on the exams. Also, there were no differences in drop rate between the two sections.

Student questionnaires and interviews indicated several possible reasons for the poorer performance by the PLATO group. Several of these were: (a) lessons not available because of lack of memory to load them, (b) errors in lessons, and (c) lessons too long. In addition, Esther Steinberg from the PLATO PEER group reviewed many of the lessons and reported that some of them were lacking in student interaction. The first problem was caused by a lack of ECS which essentially prevented students from making up lessons at odd hours or even from reviewing (or going ahead to) other lessons during scheduled hours. These problems were alleviated by the addition of more memory in January, 1975. In response to the other problems, many lessons were corrected and revised, and a few have become the object of experiments to determine what makes a good PLATO computer science lesson.

(2). In Spring 1975, CS 105 (a class of 600 freshmen in the College of Commerce and Business Administration) students spent about 90 minutes a week on PLATO. Fifty minutes replaced a lecture, and 40 minutes were spent using the interactive compiler to solve a small programming problem. Although there was no course-wide experiment conducted, the results were more positive than the previous one. Performance on the first two hour exams seemed to indicate that students learned more than they had in past CS 105 courses. (This was based on the assumption that exams from one semester to another are roughly equivalent.) These results are in spite of the fact that PLATO was down for essentially the first ten days of the semester due to hardware problems. In order to obtain a more valid measure of student achievement, the final exam was a modified version of the exam used the previous semester. For several reasons this was thought to be a safe procedure, in that students would not have access to the earlier exam. These reasons were: (1) final exams are not returned; (2) the exams were not placed in the library; (3) exams had never previously been reused; and (4) the exam was modified so that answers and details were different, even though the level of difficulty was not altered. The means on the final exam for the two semesters were nearly identical. This was a further indication that the PLATO students were learning as much as non-PLATO students.

A questionniare handed out by Esther Steinberg to 75 students on April 22, 1975, showed that most students were happy with the lessons and found them helpful.

(3).. Current experiments on individual lessons being carried out in cooperation with CERL's evaluation group are continuing in CS 103 this fall. In order to determine which teaching strategies are most useful on PLATO, comparisons will be made by implementing different versions of the same lesson. Some of the variables to be tested are: quantity of exercises, whether exercises are optional or required, and whether exercises should be scattered throughout the lesson or occur at the end. Preliminary findings indicate that while students found more frequent placement of exercises make a lesson more interesting, their performance on a quiz was not better than that of students who didn't get the extra exercises.

4. Activities: July, 1975 - June, 1976.

4.1 The library of lessons

Our library has expanded to about 135 lessons. The emphasis this year has been on making improvements to certain of these lessons, primarily the FORTRAN lessons, for use in our own courses. Some lessons, most notably fortdo, have been completely rewritten after usage experience has shown that the initial effort was completely unsatisfactory.

Among the more significant new additions to the library are a small series of lesssons on System Programming, written by or under the direction of Axel T. Schreiner. Several of these were used by Schreiner during the Spring 1976 offering of CS 323, Operating Systems. These lessons are primarily "lab" lessons, in which the students can set up and run a small system (e.g., a dispatcher using semaphores for task syncronization), and then watch the system run.

4.2 Computer assisted programming system (CAPS)

CAPS is a highly interactive diagnostic compiler/interpreter that allows beginning programmers to prepare, debug and execute fairly simple programs at a PLATO terminal. Complete syntax checking and most semantic analysis is performed as the program is entered and as it is subsequently edited. Analysis is performed character-by-character.

The most remarkable feature of CAPS is its ability to automatically diagnose errors both at compile-time and at run-time. Errors are not automatically corrected. Instead, CAPS interacts with the student to help him find the cause of his error.

Under CAPS many languages (e.g., FORTRAN, PL/1 and COBOL) have been implemented and some have ~~been~~ used in elementary programming language courses. These implementations have shown that CAPS works well in many respects and that the design aims of the system are perfectly adapted to the educational environment. Unfortunately, in the aspect of real time peformance, CAPS falls well below acceptable levels. Therefore, the development effort in the past year has concentrated on improving the performance of CAPS both during program entry and during program execution.

To understand the development let's review the organization of the compilers.

CAPS compiler organizations

Each compiler in the CAPS system consists of interpretive tables specific to the language being compiled; common driving routines to interpret these tables; and a few routines, specific to the language, that are called from the interpreted tables. These tables are built from assembler-like source code written by a compiler implementor. After generation, these tables are stored in common where they are loaded into nc variables as needed in compiling student programs.

Flow of control in the CAPS compilers is shown in Figure 1. The editor looks at each keypress the student enters from the terminal. If the key indicates a text editing function, it is performed by the editor. If the student is entering new text, each keypress is passed on to the lexical analyzer. When the lexical analyzer receives a complete token, that

26

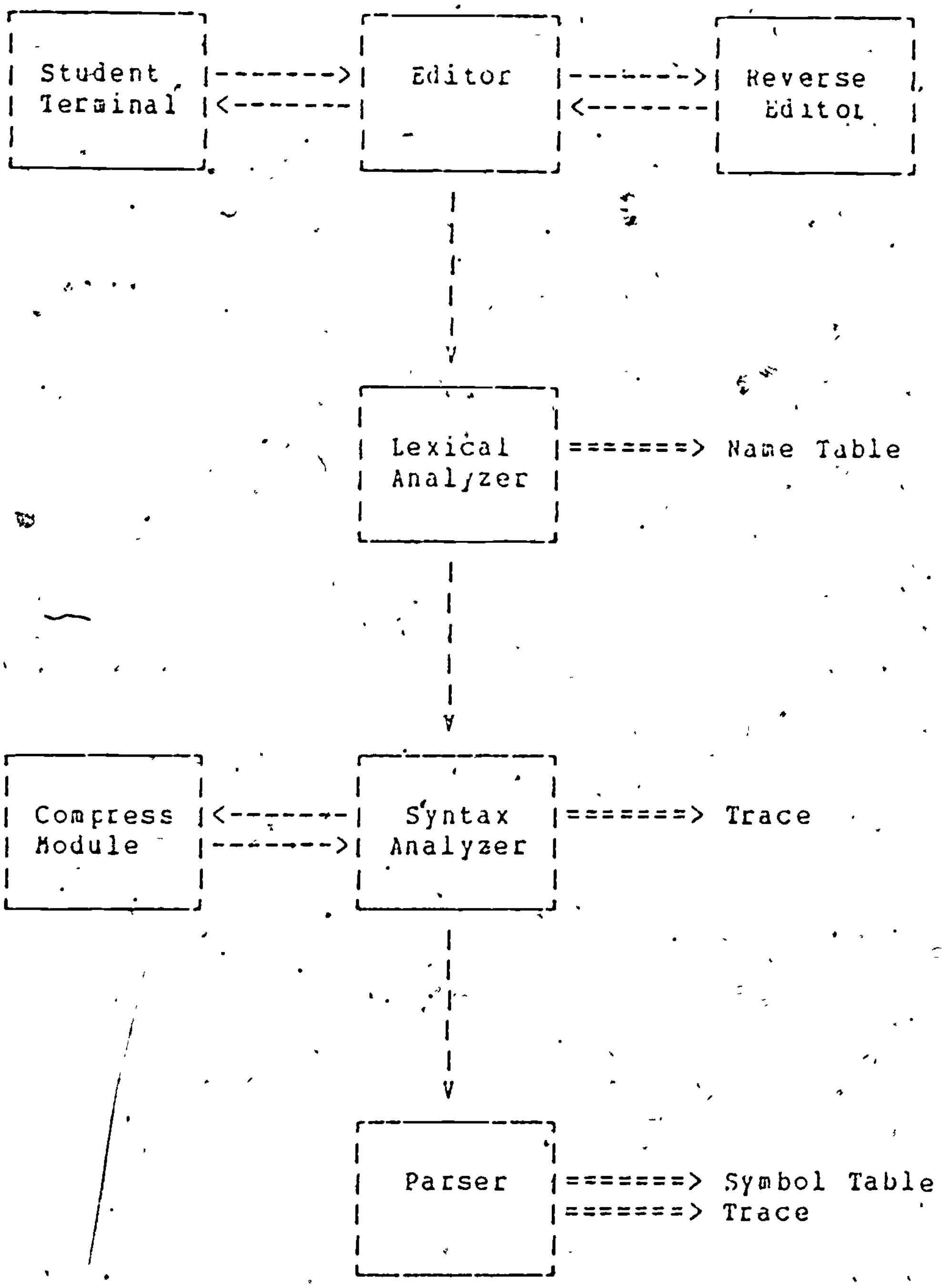

Figure 1: CAPS Compiler Modules.

token is passed on to the syntax analyzer and parser for compilation. Since each keypress is processed as it is entered, the compiler can give immediate error messages when the student enters an invalid language construct.

While compiling new text, "Trace" information is stored, allowing the Reverse Editor to uncompile the student's program as the student backs up to make a change. Occasionally the storage area for Trace information gets full. When this happens, a compression unit is called which removes alternate entries from the Trace table. After compression is performed, the reverse editor can only back up to alternate tokens. If necessary, it will back up to the previous token and then forward compile to the current token. In practice, the compression routine may be called three or four times for a student program. After four calls, there is Trace information for one out of every 16 of the first tokens entered. Closer to the "cursor" where the student is working, the Trace information is available for every token, or at least alternate tokens.

The lexical analyzer and the parser are both table driven. The table for the lexical analyzer is a state transition diagram interpreted by TUTOR code. For example, if currently in the '<' node of the diagram for PL/1, a following '=' causes transition to another node, while a following '<' or '>' is an error noted by the lexical analyzer. Conversely, in BASIC '< >' is the "not equal" token.

The tables in the Parser are not just a state transition array, as in the lexical analyzer, but consist of internal codes interpreted by a TUTOR unit in each compiler. This internal code is complete with arithmetic operations, conditional jumps, calls to error routines, and calls to the lexical analyzer to receive the next token. Thus the student writes in PL/1, for example, and his program is compiled by code being interpreted by TUTOR code being interpreted by PLATO run time routines.

```
            CM                                     ECS

 nc variables (1500)                    Storage          (644)
 +---------------------------+          +---------------------------+
 |                           |          |   Parse Storage 53        |
 |     Lexical               |          +---------------------------+
 |       or                  |          |v  Symbol Table 109        |
 |     Parse                 |          +---------------------------+
 |     Tables                |          |v  Name Table     64       |
 |                           |          +---------------------------+
 |     400                   |          |v  Char Table    168       |
 |                           |          |                           |
 +---------------------------+          +---------------------------+
 |     Parse Storage 53      |          |   Hash Table     20       |
 +---------------------------+          +---------------------------+
 |v    Symbol Table 109      |          |   Text           60       |
 +---------------------------+          +---------------------------+
 |c    Symbol Table 210      |          |   Trace          98       |
 |                           |          +---------------------------+
 +---------------------------+          |   Variables      88       |
 |c    Name Table    110     |          +---------------------------+
 +---------------------------+
 |v    Name Table     64     |          Common          (1286)
 +---------------------------+          +---------------------------+
 |v    Char Table    168     |          |                           |
 |                           |          |   Parse                   |
 +---------------------------+          |   Table                   |
 |c    Char Table    119     |          |                           |
 |                           |          |   400                     |
 +---------------------------+          |                           |
 |     Hash Table     20     |          |                           |
 +---------------------------+          +---------------------------+
 |     Text           60     |          |                           |
 +---------------------------+          |   Lexical                 |
 |     Trace          98     |          |   Table                   |
 +---------------------------+          |                           |
 |     Variables      88     |          |   400                     |
 +---------------------------+          |                           |
                                        +---------------------------+
                                        |   Pointers        22      |
                                        +---------------------------+
                                        |c  Symbol Table 210        |
 v = variable portion                   |                           |
 c = constant portion                   |                           |
 number = length of table               +---------------------------+
                                        |c  Name Table     110      |
                                        +---------------------------+
                                        |c  Char Table     119      |
                                        +---------------------------+
```

Figure 2: CAPS Data Areas

Compiler data structures

The CAPS compilers use "common" for pointers and tables shared by all users of one compiler and "storage" for all pointers and tables needed by an individual user. Few, if any, of the student variables are used by the compilers. Portions of common and storage may be loaded into 1500 'nc' variables in central memory. However, at most three areas of each may be loaded at once. As shown in Figure 2, by arranging the data areas in ECS carefully, it was possible to meet this three-area restriction and still get the tables in desired locations in central memory. However, the lexical and parse tables are each 400 words long, and only one of them can be loaded at once. (This is significant since the compilers spend 8% of their time changing the loading arrangement.) Figure 2 shows the layout of these areas.

Possible improvements

Four suggestions for improving the CPU time requirements of the editor have been made. The first involves minor recoding of critical sections of the compiler and would give a minor improvement in speed. Two such changes are to 1) stop displaying the "space left" indicator on the student's screen -- 2.5% improvement, and 2) stop doing unnecessary -stoload- commands--2% speedup. (The compiler is reexecuting the same -stoload- command once for every token, which is unnecessary, at least for the PL/1 compiler.)

The second suggestion involves recoding the lexical analyzer or parser in TUTOR, rather than having TUTOR code interpret these tables. This would give an unknown amount of speedup, estimated at 20% for the lexical analyzer, more for the parser, but reduces the generality of the driver program.

30

The third method proposed for speeding up the editor is to process one line of source text at a time rather than one character at a time. This moves the collection process from the TUTOR lesson to the PLATO system, and allows the user to correct errors in his current line before giving it to the lexical analyzer and parser. This method has been implemented and tested, and does reduce the CPU time used, but the system is then unable to respond immediately to an invalid character or token.

The fourth method suggested is to move semantic checking and symbol table construction to a later pass. Timing tests conducted by L. White suggest anticipated improvement is probably more than 20%.

Of the four possible improvements, this last is least objectionable since it retains the table-driven nature of the CAPS editor and still permits instantaneous analysis of many errors commonly made by the programming neophyte. In addition, the second compiler pass permits generation of a representation of the program that is more easily interpreted so that execution speed is improved as well.

Work in progress.

In the past year, S. Nakamura has implemented an experimental FORTRAN compiler using the two pass organization with encouraging results. His compiler is diagrammed in Figure 3.

Paralleling Nakamura's experimental compiler development is an effort by T. Fishman to improve the parser module of the editor. The current parsing tables must be programmed by the compiler-writer. Considerable knowledge and effort is required to implement a language and the process is prone to error. Since, in the reorganized editor, the parser is responsible only for syntactic details, more formal, grammar-driven parsing methods are possible. In particular, Fishman is investigating the possibility of using an

31

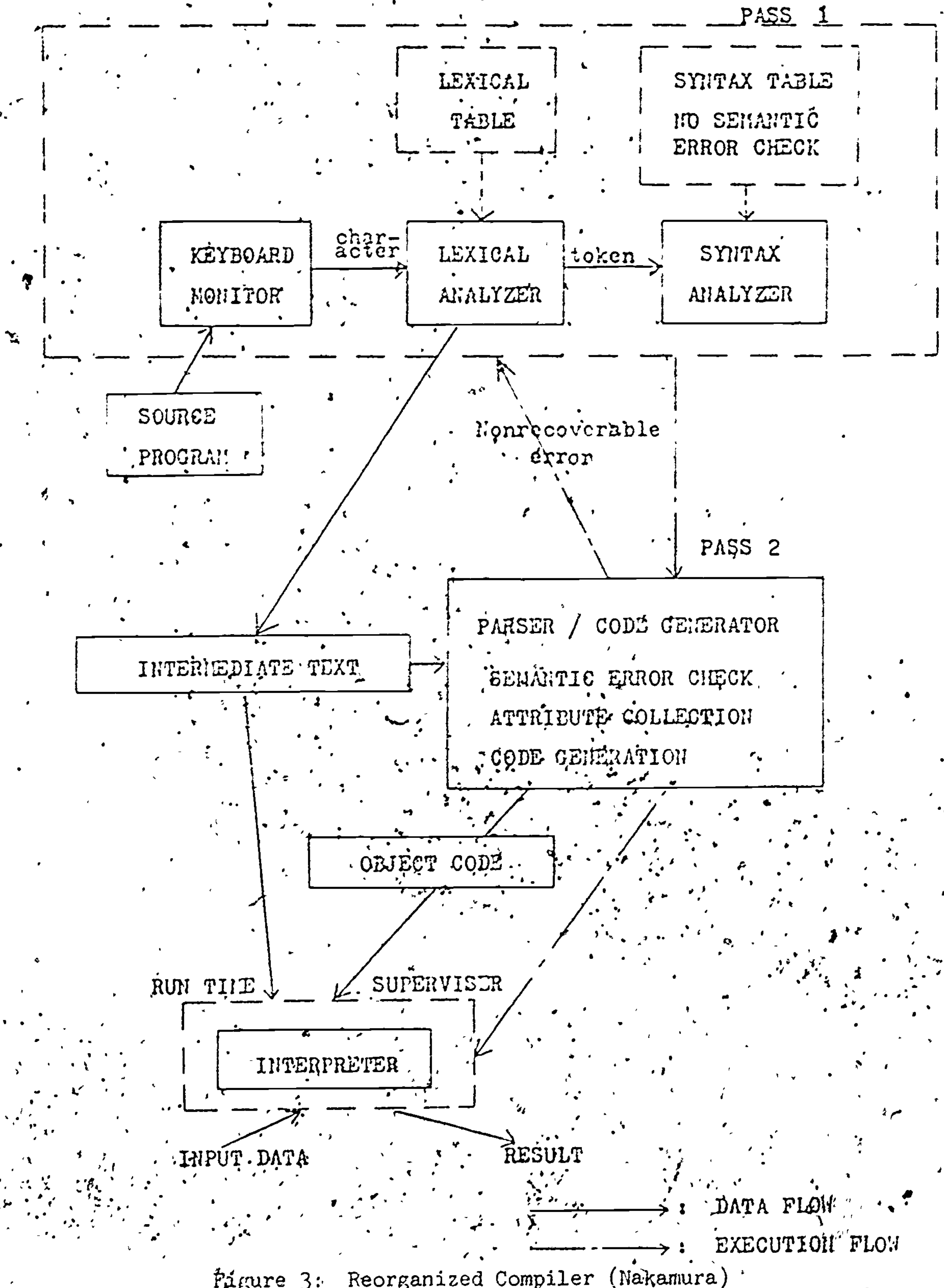

Figure 3: Reorganized Compiler (Nakamura)

32

LALR parsing technique in CAPS. The advantages of an LALR parser is that both the parser and the parsing table will be more compact than the current parser and the table can be constructed directly from the formal grammar specifying the programming language.

The output of the new parser will have more of the syntactic structure of the program built into the intermediate text, which will permit the editor to deal directly with syntactic units such as statements and expressions. As long as these syntactic units are left intact during editing, the parser will not have to be called to unparse them as the cursor moves past.

The structured intermediate text should also permit the second pass to be table-driven to a greater extent that it is now. B. Speelpenning is currently revising Nakamura's FORTRAN compiler to use the output of Fishman's LALR parser to make use of this structure.

Conclusion

The attempts by Nakamura, Fishman, and Speelpenning are essential to the continued existence of CAPS at its current level of diagnostic assistance. Experience has shown that in its original formulation, CAPS would be intolerably slow in rigorous classroom use and this degraded performance is directly a consequence of providing enhanced diagnostic assistance. So should the current research prove fruitless, we must conclude that at the level of CPU processing currently available, the desired level of automatic error diagnosis is unattainable on PLATO.

4.3 Exam system

The Generative Exam System developed by L. Whitlock is a completely interactive system for construction and administration of examinations. During a single terminal session, the system can administer an examination, grade it, and allow the student to compare his answers with the correct ones.

Examination problems are generated by the system according to specifications written by an instructor. Analyses of student performance, class performance, and examinations are also provided by the system.

Figure 4 is a block diagram of the major components of the Generative Exam System. All users enter the system through the Monitor. Students take and review exams in the Exam Administration section. Instructors write exams in the Exam Writing section and work with exam results in the Exam Statistics section. Problem Generator/Graders (pg/g) provide the problems for the exams. Each pg/g handles all aspects of a problem except data storage; it assists an instructor in writing problem specifications, generates a problem for each student, administers and grades that problem, reviews it with the student, and collects data for improving future problems.

When a user first enters the Exam System, he is allocated a record in the Student Records data area and a permanent storage area for his work in the Student Exams data area. The system differentiates between two kinds of users--student and instructor. An instructor has access to both student and instructor options while all other users have access to the student options only. An instructor writes an exam by writing problem specifications for each desired pg/g. This set of problem specifications is assembled into an exam specification and stored in the Exam Specs data area. When a student takes an exam, the appropriate exam specification is transferred from the Exam Specs data base to the student's permanent storage area in the Student Exams data base. The same area is used to record his work as he changes from problem to problem.

This past year pg/g's have been written for many aspects of FORTRAN. With these, various versions of the system have been used to give not-for-credit exams in several introductory computer science courses, and part of

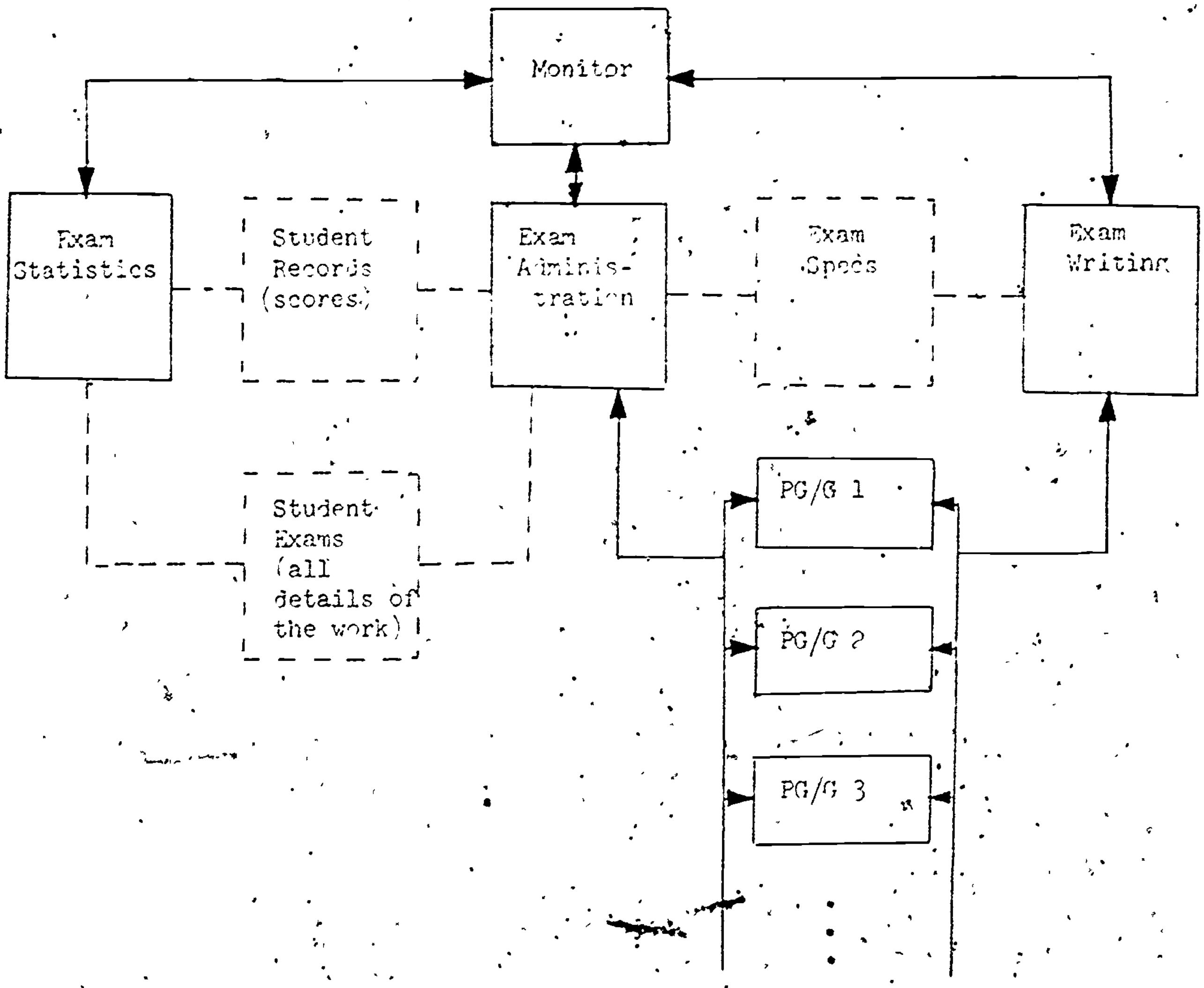

Figure 4: BLOCK DIAGRAM OF THE MAJOR COMPONENTS OF THE GENERATIVE EXAM SYSTEM.

for-credit examinations in two courses (CS 101 and CS 103). The most popular (with students) use of the system has been to provide practice examinations just prior to paper examinations; students may take these as often as they like.

During early experiments with the system it was noticed that students were spending roughly twice as long on PLATO exams as they would on an equivalent paper exam. Analysis of video tape of four students taking an exam on PLATO and on paper revealed that 40% of the time was due to quirks of the system we have since eliminated; 25% is an unavoidable artifact (mostly terminal speed); and about 35% was due simply to longer think-time. This think-time may result from an unconscious expectation of an immediate, irreversible grade; though this is not the case in the exam system, it is usual in other instructional uses of PLATO.

Our major experimental effort of the past year has been an exploration of alternate styles of examination, based on a set of pg/g's which can generate problems to a specified "difficulty level". In a "gambling" style the student chooses a difficulty level indirectly by specifying, for each problem, what percentage of the maximum points he would like to try for. In a "tailored" style, the system adjusts the difficulty level on the basis of the student's prior performance on the problem. At a minimum, the tailored style eliminates the problems attendant on giving an examination that is too hard or too easy.

Initial results show that the tailored exam subjects got more points than any other group, even though maximum points can only be achieved by correctly working a problem at the highest difficulty level. In response to a questionnaire, students showed preference for the tailored style. Experiments are continuing.

In an effort related to the exam system, a special quiz system has been developed which enables presentation of a criterion-referenced quiz following a PLATO computer science lesson. Designed and implemented by R. Anderson from a concept proposed by R. Montanelli, the system consists of a central system monitor and numerous "quiz generator/graders", each of the latter generating quiz questions and grading them in a manner identical to that of the problem generator/graders of the exam system.

The system provides a means to:

- assess the completeness and effectiveness of instruction within a PLATO CS lesson. Each quiz is developed from the objectives which produced the CS lesson which it follows, thus student responses accumulated by a quiz should identify lesson errors and deficiencies as well as weaknesses in the quiz itself.
- facilitate student learning of lesson content. Alterations to both the lesson and the quiz based on the analysis of quiz question responses should yield a quiz that will aid a student in determining what has and hasn't been learned from the material fully covered by the lesson.

An initial trial of the system occurred during the fall semester of 1975 when one quiz was presented following its corresponding lesson. Responses accumulated for each quiz question clearly indicated lesson shortcomings which were later eliminated via additions to and the restructuring of the lesson.

With the development of more quiz generator/graders, further trials were attempted during spring semester of 1976. An experiment to determine student performance tendencies on different types of quizzes was also conducted. Data accumulated by the quizzes and the experiment are currently being analyzed.

4.4 Automatic judging of student programs

An automated system for instruction should be capable of making judgements and providing comments on student programs, analogous to the role played by teaching assistants and graders in the more traditional means of instruction. Our efforts to provide this capability have resulted in i) two lessons which ask the student to write fairly sophisticated programs and attempt to judge these programs interactively with respect to both correctness and good design, and ii) the development of technques capable of automatically detecting anomalies in beginning student's programs without knowledge of the user algorithm being implemented.

A program by R. Danielson, which exposes students to a dynamic example of the top-down programming process by monitoring their attempts to write a PL/1 program for symbolic differentiation of a polynomial, has been completed. PATTIE (Programmed Aid for Teaching Top-down programming by Interactive Example) mimics the action of a human tutor, in that she engages the student in an interactive dialog, judging the correctness of student-suggested refinements and providing hints and comments where necessary.

The tutor uses an AND-OR graph as a model of the stepwise refinement process, which student and tutor traverse together in the course of program development. The interactive dialog is conducted in natural language, parsed via a keyword/pattern matching scheme which is built into the PLATO author language, TUTOR. This simple scheme is effective (80% of inputs understood) due to the limited domain of discourse. A small amount of testing of PATTIE with beginning students has been conducted, and more is planned for the immediate future.

The other lesson is a sorting laboratory and program verification system developed by P. Mateti. This system allows the student to write an arbitrary in-place sorting program in a programming language with specially designed sorting primitives. A special interpreter then provides a dynamic display of the status of the array and indices during execution. This laboratory is an excellent facility for understanding the operation of various sorting algorithms, and experimenting with different implementations of one particular algorithm.

In addition, the student may provide assertions about the state of the keys in the array; and the truth or falsity of these assertions is indicated during execution. The student may submit completed programs to the program verification routines, which use the inductive assertion method to prove the program correct, or prove it incorrect and provide a counter-example. A special theorem prover, which is highly efficient in this restricted domain, is the heart of the system. Unfortunately, while the verifier can prove or disprove a sorting program faster than any other known program (for example, a bubble sort routine is proven in approximately nine CPU-seconds), the severe limitations on processor use imposed by PLATO make use of the verifier for actual instruction difficult (the same bubble sort may require as much as 30 clock-minutes to prove).

A third project is being conducted by W. Gillett to automatically detect program anomalies (in procedural languages such as FORTRAN or PL/1) without knowledge of the user algorithm being implemented. Data has been collected and is currently being analyzed to identify and categorize programming "defects" made by beginning programmers. Many of these "defects" occur because the student views his program as a sequence of essentially independent statements instead of an integrated whole.

Detection techniques based on iterative global flow algorithms have been developed. Even though the user algorithm being implemented is unknown, these techniques are capable of describing the "defect" in detail and can, in many cases, direct the student toward a resolution of the "defect".

4.5 The KAIL project

D. Embley explored experimental and formal language design and applied these design methods to an investigation of control constructs for interactive computing, particularly in computer-aided instruction. As a means for exploration, a new CAI author language, KAIL, was designed. One feature, the KAIL selector [Embley and Hansen 76], handles structured flow in interactive dialogues. This construct not only unifies selection and iteration, it subsumes CAI answer judging as well. For nonstructured flow of control, KAIL includes a static exception processing scheme somewhat similar to PL/1 on-conditions.

These control constructs were tested in two experiments conducted on-line in the PLATO environment. The first [Embley 75] matched the KAIL selector against a set of typical ALGOL-like constructs. The results indicate that subjects understood programs written using the KAIL selector at least as well as and perhaps better than programs written using typical <u>if</u>, <u>case</u>, and <u>while</u> constructs. The second experiment [Embley 76] utilized the PLATO system to monitor subjects as they found and fixed program bugs and modified a substantial CAI lesson about 500 lines in length. Some subjects saw a version written in a TUTOR-like language, others saw a version in a KAIL-like language. Observations basically supported the KAIL sequencing constructs, but also uncovered unforeseen difficulties in the KAIL-like language. The results of the two experiments generally indicate that the KAIL constructs are likely to be psychologically sound.

These constructs were also examined through a formal definition of their semantics. An axiomatic approach was applied to the KAIL selector, and one suggested extension was found to be an order of magnitude more "complex" than the basic selector construct. Both the KAIL and the TUTOR exception processing schemes were defined in terms of a behavioral model, and the two schemes were compared. The formalism exposed context dependencies in TUTOR and showed why programmers are likely to find the TUTOR sequencing constructs more difficult to understand and use than the corresponding constructs in KAIL.

As a result of the investigation, three basic design principles evolved: uniformity, separability, and locality. The uniformity principle suggests that languages ought to be designed with a one-to-one relationship between syntax and semantics. The separability principle suggests that special-purpose, composite structures that are only indirectly separable may be harmful. The locality principle suggests that language features should be as permanent and local as possible. These three principles are proposed as a partial basis for the design of programming language features in general.

4.6 Use of ACSES in instruction

PLATO is now in routine use in CS 101, 102, 103, 105, and 400 for a total of 1000-1200 students per semester on the U of I campus. In fall 1976, CS 101 and 105 will each have one less professor assigned, an indication of the acceptance of PLATO. Other introductory courses are in the process of phasing in PLATO.

Also, there has been systematic use of ACSES lesson materials in six sections of three courses at Wright Community College in Chicago, mainly under the direction of Kathleen Galway. The table below summarizes this use for the fall 1975 semester.

	Students	Hours per student	Session/student	Course Description
DP 101	35	2.8	7	Introduction
DP 101	27	3.2	8	Introduction
DP 101	34	2.2	7	Introduction
DP 106	33	8.2	14	FORTRAN
DP 106	23	5.8	14	FORTRAN
DP 135	24	1.1	3	COBOL

As can easily be seen, the heaviest use has been in the FORTRAN course (DP 106), mainly because we have spent more time developing FORTRAN lessons.

In order to assess the effect of replacing one lecture a week with a PLATO lesson in a large university class, a controlled experiment (described below) was run in fall 1975. Essentially the results indicated that PLATO students would strongly recommend PLATO sections to their friends, learned as much as non-PLATO students, but dropped at a somewhat higher rate.

(1). <u>Procedure</u>

In order to allow for testing some hypotheses about the use of PLATO in our introductory computer science courses, a controlled experiment was designed for CS 105, for fall 1975. Five lecture sections were offered, with four of them arranged in the following way. Professor A taught a PLATO section (one hour on PLATO replacing one lecture) at 9:00 am, and a non-PLATO section (two lectures, no access to PLATO) at 10:00 am. Professor B taught a non-PLATO section at 9:00 am and a PLATO section at 10:00. Professor C, who was in charge of the course, taught a fifth (PLATO) section in the afternoon, but it was not involved in the experiment. It should be noted that this author did not teach any sections, and that neither professor A nor B had ever used PLATO before the semester. An additional minor difference between the two types of sections was that the PLATO students did their first two machine problems using on-line PLATO FORTRAN compilers instead of the IBM 360.

Before the semester began, students' registrations for the five sections were equalized (by the computer program which makes up student schedules), so that there were equal numbers of students in each section. Then students in the 9:00 and 10:00 sections were randomly reassigned to either a PLATO or non-PLATO section. Students were kept separate by using the following mechanisms. 1) Students were not allowed to transfer sections unless it could be done without affecting the experiment. For example, a student could not stay at the same hour and change sections. If a student wanted to change from 9:00 to 10:00 (or vice-versa), he/she had to keep the same type of section (PLATO or non-PLATO). If a student wanted to switch from the afternoon section to a morning section, he/she was randomly assigned to a PLATO or non-PLATO section. Students were not allowed to transfer to the afternoon section. Any student who felt she/he could not abide by these rules was sent to the author who attempted to convince her/him of the value of educational experiments and of PLATO (There were about ten such students, and most wanted out of PLATO, presumably due to having heard about the problems from the previous semester). The result of these discussions was that two students who protested violently against PLATO and machines in general were allowed to transfer out of PLATO sections.
2) Students in non-PLATO sections were not given access to PLATO, as a student record with associated name and code had to be created for each PLATO student, and this was not done for non-PLATO students. 3) Individual attendance was taken in the lectures intended for non-PLATO students only, insuring that no

PLATO students could enter. In order to assess the effect of these precautions, students were asked (via a questionnaire given to all students immediately after completing their CS 105 final exams) if they had looked at materials for the other group (they were assured that there would be no penalty). A few non-PLATO students had seen some of the PLATO materials through friends or through having access through other courses, and a few PLATO students attended lectures, either through using the name of a friend who was not attending or through slipups in our records. It was felt that none of these minor disturbances would have any major effect on results.

The three hypotheses of this study were:

1. PLATO students would enjoy the course more, and give it a stronger recommendation to their friends.
2. PLATO and non-PLATO students would perform equally well on exams and homeworks in the course.
3. The drop rates in the two types of sections would be similar.

(2). Results

In answer to the question (from the questionnaire administered with the final exam): 'If a friend were taking CS 105 next spring and PLATO and non-PLATO sections were offered, what would you recommend he take?' PLATO students strongly recommended PLATO (112 circled 'definitely PLATO', 88 'PLATO if convenient', 45 had 'no recommendation', 15 said 'lecture if convenient', and 21 said 'definitely lecture'. On the other hand, non-PLATO students were neutral (their responses, in order, were 29, 22, 91, 26, 19), or even showed a slight preference for PLATO.

In order to compare learning across groups, a 2x2 univariate analysis of variance was computed for each exam and for total points on

44

computer programs. No significant differences were found, and means were nearly identical for the various groups (Table 1).

The third hypothesis concerning drop rates was rejected, however. Professor A had 19 (15%) drops from his PLATO section, and only 4 (4%) from non-PLATO. Professor B had 28 (25%) drops from PLATO, and 18 (14%) from non-PLATO.

Table 1
Average Scores on Exams
and Machine Problems

(a) Machine Problems

		PLATO	non-PLATO
Professor	A	143	142
	B	140	146

(b) Hour Exam 1

Professor	A	92	91
	B	91	91

(c) Hour Exam 2

Professor	A	65	62
	B	62	62

(d) Final Exam

Professor	A	140	134
	B	135	134

(3). <u>Discussion</u>

Students in the PLATO groups would strongly recommend that their friends choose PLATO sections, thus confirming the first hypothesis. Even if the 'extra' 25 drops in the PLATO sections were strongly negative, they would have a small effect on the totals of [illegible] PLATO students recommending PLATO, and only 36 of them recommending lectures. It should be remarked that when the PLATO students were asked to indicate what they thought were the worst features of PLATO, 78 checked 'The distance to CERL' (Unfortunately the terminals are located on the north edge of campus in CERL, about a mile from most commerce courses.), while 37 checked 'Lack of human contact', and 31 checked 'PLATO going down', the next two most frequently checked responses. Thus the major problem was unfortunately out of our control.

The second hypothesis was not rejected. The nearly identical scores on exams and machine problems in Table 1 are evidence of that, without reference to statistics. There is no reason to suspect that the PLATO drops were poor students. However, if the dropped PLATO students were below average, they could not have had a large enough effect on the results to alter the obvious conclusion. This result is certainly in agreement with most studies of the effects of CAI. In fact, when Jamison, Suppes, and Wells (1974) surveyed the effectiveness of alternative instructional media, they stated:

'... the equal-effectiveness conclusion seems to be broadly correct for most alternate methods of instruction at the college level ...',

and suggested studying costs of various methods of delivery. However, a major advantage of CAI is that once used, it is not set in stone like a textbook or movie. As a result of this experiment, the two lessons which students liked the least are being rewritten from scratch. Secondly, a quiz system has been begun. When completed, it will present a quiz to each student at the completion of each lesson. The quizzes are not written by the authors of the lessons, and in fact quiz authors are discouraged from looking at the lessons. However, the quizzes are written from the same objectives that were used to write the lessons. The resulting quiz scores will not only tell the students how well they understand the material which the lesson is supposed to cover, but will tell instructors and lesson authors how well the lesson is working. Thus, continual improvement is possible, and perhaps eventually CAI materials will be as good as the best lecturer, and therefore better than many.

On the other hand, the hypothesis about equal drop rates was rejected. This was a surprising result, especially when the smaller experiment a year earlier (under worse conditions) showed no differences. However, the earlier course may have been a special case. It was a relatively small elective course with mainly juniors and seniors in psychology and similar fields. These students were more involved and interested in the experiment (as the author was teaching the course), and they may have stayed for that reason. On the other hand, CS 105 is a required course for freshmen in the college of commerce, and the students were presumably less interested in long term educational goals (for themselves as well as for the PLATO materials). However, although this drop rate was disturbing, there were a few likely reasons for it, all of which could be fixed. For one thing, the first three weeks were confusing for the students because they had pre-enrolled in a course which they expected would consist of two lectures and a discussion each week. Instead, three-fifths of them had a lecture cancelled and had to sign up for a PLATO section instead. These sections caused a lot of trouble, as some were scheduled for week-ends, and many students complained that they were unable to meet any of the remaining available PLATO times. Although most of this confusion was necessary due to the nature of the experiment, in the future students will preregister for PLATO sections just as for any other class. A second possible cause for the different drop rates was that for the first few weeks, PLATO students were required to do their programming problems in one of the on-line, interactive compilers. Although it was thought that this would be fun for the students, the compiler gave very poor response time because of the amount of processing going on to check for errors after each student keypress. Finally, drops might have been due in part to student dissatisfaction with the two poor lessons which were later rewritten. Students had not been systematically polled about the lessons before, and the relatively negative reaction to two of them was quite surprising.

Another possible explanation for the higher drop rate on PLATO, is that some students (< 10%) are anti-machine and that CAI will always have this problem. The author does not feel that the large differences found here could be attributed to this reason. However, this is being checked during the current semester, because the problems mentioned in the preceding paragraph have been fixed, and although no experiment is being run during the current semester (all five sections of CS 105 are using PLATO), the current drop rate could be compared with past drop rates, in order to see if the PLATO drop rate is as high as was found here.

(4). Summary

PLATO lessons can be used to replace one lecture a week in an introductory computer programming course. Students learn as much and prefer PLATO to large lecture sections. The remaining problems are: 1) Is there a higher drop rate on PLATO? and 2) Can instruction be improved through continued development of the CAI materials?

5. Papers and Reports

Anderson, R. I., "User's manual and guide to the ACSYS quiz system," to appear as DCS Report, September 1976.

Anderson, R. I., "An experiment on modes of question answering", to appear as DCS Report, September 1976.

Barber, J. A., "Data collection as an improvement technique for PLATO lessons", Report UIUCDCS-R-75-777 (M.S. Thesis), Department of Computer Science, University of Illinois, December 1975.

Barnett, R. D., "An interactive COBOL system for PLATO", Report UIUCDCS-R-75-685 (M.S. Thesis), Department of Computer Science, University of Illinois, January 1975.

Danielson, R. and Nievergelt, J. (1975), "An automatic tutor for introductory programming students", Proc. Fifth Symp. on Computer Science Education, SIGCSE Bulletin, Vol. 7, No. 1, February 1975.

Danielson, R. L., "PATTIE: An automated tutor for top-down programming", Report UIUCDCS-R-75-753 (Ph.D. Thesis), Department of Computer Science, University of Illinois, October 1975.

Davis, A., Tindall, M. H. and Wilcox, T. R. (1975), "interactive error diagnostics for an instructional programming system", Proc. Fifth Symp. on Computer Science Education, SIGCSE Bulletin, Vol. 7, No. 1, February 1975.

Davis, A. M., "An interactive analysis system for execution-time errors", Report UIUCDCS-R-75-695 (Ph.D. Thesis), Department of Computer Science, University of Illinois, January 1975.

Eland, D. R., "An information and advising system for an automated introductory computer science course", Report UIUCDCS-R-75-738 (Ph.D. Thesis), Department of Computer Science, University of Illinois, June 1975.

Embley, D. W., "An experiment on a unified control construct", Report UIUCDCS-R-75-759, Department of Computer Science, University of Illinois, October 1975.

Embley, D. W. and Hansen, W. J., "The KAIL selector - a unified control construct", SIGPLAN Notices, Vol. 11, No. 1, January 1976.

Embley, D. W., "An experiment on CAI sequencing constructs", Report UIUCDCS-R-76-771, Department of Computer Science, University of Illinois, February 1976.

Embley, D. W., "Experimental and formal language design applied to control constructs for interactive computing" (Ph.D. Thesis), to appear as DCS Report, July 1976.

Gillett, W. D., "An interactive program advising system", Proceedings of SIGCSE-SIGCUE Joint Symposium on Computer Science Education, SIGCSE Bulletin, Vol. 8, No. 1, February 1976.

Gillett, W. D., "Iterative techniques for detection of program anomalies", submitted to the Conference on Principles of Programming Languages, Los Angeles, California, January 1977.

Gillett, W. D., "Interval maintenance in an interactive environment", in preparation.

Izquierdo, F. J., "A generator/grader of problems about syntax of programming languages to be used in an automated exam system", Report UIUCDCS-R-75-755 (M.S. Thesis), Deparment of Computer Science, University of Illinois, September 1975.

Mateti, P., "An automatic verifièr for a class of sorting programs" (Ph.D. Thesis), to appear as DCS Report, September 1976.

Montanelli, R. G., Jr., "CS 103 PLATO experiment, Fall 1974", Report UIUCDCS-R-75-746, Department of Computer Science, University of Illinois, July 1975.

Montanelli, R. G., Jr., "Evaluation of the use of CAI materials in an introductory computer science course", presented at the AEDS International Convention, Phoenix, Arizona, May 1976.

Montanelli, R. G., Jr., "Using CAI to teach introductory computer programming", submitted to Communications of the ACM.

Montanelli, R. G., Jr. and Steinberg, E. R., "Using PLATO to teach introductory computer science - an overall evaluation," in preparation.

Nakamura, S., "Reorganization of an interactive compiler" (M.S. Thesis), to appear as DCS Report, August 1976.

Nievergelt, J., Reingold, E. M. and Wilcox, T. R., "The automation of introductory computer science courses", in A. Günther, et al. (eds), International Computing Symposium 1973, North-Holland Publishing Co., 1974.

Nievergelt, J., et al., "ACSES, an automated computer science education system", Angewandte Informatik, Vol. 3, April 1975.

Nievergelt, J., "Interactive systems for education -- the new look of CAI", Invited paper presented at the IFIP World Conf. on Computer Education, Marselle, France, September 1975.

Nievergelt, J., et al., "ACSES: the automated computer science education system at the University of Illinois", Report UIUCDCS-R-76-810, Department of Computer Science, University of Illinois, August 1976.

Pradels, J. L., "The Guide, an information system", Report UIUCDCS-R-74-626 (Ph.D. Thesis), Department of Computer Science, University of Illinois, March 1974.

Renshaw, J. W., "Robocar: educating the layman in computer science", Report UIUCDCS-R-75-741 (M. S. Thesis), Department of Computer Science, University of Illinois, July 1975.

Segal, B. Z., "A comparison of student performance under two methods of error announcement", Report UIUCDCS-R-75-727 (M.S. Thesis), Department of Computer Science, University of Illinois, May 1975.

Steinberg, E. R. and Montanelli, R. G., Jr., "Effects of coerciveness and aspects of human-machine interaction in a computer science CAI lesson", to be submitted to Journal of Computer-based Instruction.

Tindall, M. H., "An interactive table-driven parser system", Report UIUCDCS-R-75-745 (M.S. Thesis), Department of Computer Science, University of Illinois, August 1975.

Tindall, M. H., "An interactive compile-time diagnostic system", Report UIUCDCS-R-75-748 (Ph.D. Thesis), Department of Computer Science, University of Illinois, October 1975.

White, L. A., "CAPS compiler CPU use report", Report UIUCDCS-R-75-790, Department of Computer Science, University of Illinois, December 1975.

Whitlock, L. R., "Interactive test construction and administration in the generative exam system" (Ph.D. Thesis), to appear as DCS Report, September 1976.

Wilcox, T. R., "The interactive compiler as a consultant in the computer aided instruction of programming", Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems, March 1973.

Wilcox, T. R., Davis, A. and Tindall, M. H., "The design and implementation of a table-driven, interactive diagnostic programming system", to appear in Communications of the ACM.

Wilcox, T. R., "An interactive table-driven diagnostic editor for high-level programming language", in preparation.