To the Graduate Council:

I am submitting herewith a thesis written by Alan Vincent Stone entitled "The Design, Implementation, and Analysis of a Text Expansion System for COURSEWRITER III." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Alan Davis

Alan M. Davis, Major Professor

We have read this thesis
and recommend its acceptance:

Sara R. Jordan

David Smyth

Accepted for the Council:

Hilton A. Smith

Vice Chancellor
Graduate Studies and Research

THE DESIGN, IMPLEMENTATION, AND ANALYSIS OF A TEXT EXPANSION SYSTEM FOR COURSEWRITER III

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Alan Vincent Stone

August 1976

1288092

ACKNOWLEDGMENTS

The author wishes to express his sincere appreciation and thanks to Dr. Alan M. Davis whose guidance and suggestions were instrumental in the development of this thesis. Special thanks also go to Dr. David W. Straight and Dr. Sara R. Jordan for their service on the committee. Additionally, the author would like to thank Pamela R. Hewlett for her expert job in the typing of this thesis and for coping with so many last minute changes in both content and format.

I would also like to express my deepest appreciation to my fiancée Kandy Callicott for her patience and understanding while I incessantly and needlessly explained the content of this thesis. Though the topic was meaningless to her, her support was invaluable.

Last, but not least, I would like specially to thank my parents for their continuous support and faith in me and many other things without which I am certain this thesis could never have been completed.

ABSTRACT

The purpose of this study was to develop a programmed system that would enable a lesson author to more easily create a CAI program using IBM's COURSEWRITER III Version 3 programming language. The problems considered were the difficulties and inconveniences of writing COURSEWRITER lessons incorporating questions with many similar predicted responses. Questions of this type require extensive enumeration of many possible responses and place an unnecessary burden upon the lessen author.

As a solution, the SLATE system was constructed that processes language syntax very similar to the Backus-Naur like form used in the GIZMO programming language from the University of Illinois. A single SLATE statement is actually a COURSEWRITER major operation code with text that represents many actual COURSEWRITER statements. SLATE expands the single statement and creates a file that is later inserted into COURSEWRITER system datasets. Minor operation code statements can also be condensed when specified in the SLATE input. Other conveniences are provided by the system to permit less time spent on programming and entering a lesson and more time allocated to lesson development.

The SLATE system is effective in performing its designated tasks and when used optimally can result in great savings of time to a COURSEWRITER lesson author. There are problems that exist at

this time, though. Mostly, these problems are within the COURSE-WRITER system itself. With additions to SLATE and reconstruction of parts of the COURSEWRITER system, it is possible that these problems can be solved making COURSEWRITER a useful system for computer assisted instruction.

TABLE OF CONTENTS

CHAPTER		PAGE
I.	INTRODUCTION	1
II.	CAI BACKGROUND	4
III.	CAI SYSTEMS	19
IV.	COURSEWRITER III	35
V.	SLATE: AN EXTENDED COURSEWRITER III	51
VI.	INTERNAL STRUCTURE AND OPERATION	77
VII.	CONCLUSION	93
LIST OF REFERENCES		97
APPENDIXES		101
APPENDIX A.	PL/I PROGRAM LISTING	102
APPENDIX B.	EXAMPLE USAGE OF THE SYSTEM	136
APPENDIX C.	JOB CONTROL LANGUAGE NEEDED FOR OPERATION ON THE IBM 360/65 AT UTCC	166
VITA		170

LIST OF TABLES

TABLE		PAGE
I.	COURSEWRITER Opcodes and Associated Meanings	37
II.	Parameters and Action Taken If Set to 1.	44
III.	System Action for Occurrence of Parameter Interaction	46
IV.	Listing of SLATE Keywords, Meanings, and Default Values.	68

LIST OF FIGURES

FIGURE		PAGE
V.1	Demonstration of GIZMO Lesson for Sentence Structure	54
V.2	Listing of Some of the COURSEWRITER Needed to Program Sentence Structure Question	55
V.3	SLATE Instructions to Generate Entire COURSEWRITER Code for Lesson on Sentence Structure	57
V.4	SLATE Code Using Predefined Major Expressions and Predefined Minor Opcodes.	65
VI.1	Flow Diagram of the SLATE Processing System.	78
VI.2	Procedure for Replacement of PME's in Text of SLATE Instructions.	80
VI.3	Routine ORCHK Flow Diagram	82
VI.4	Recursive Algorithm for COURSEWRITER Code Generation .	83
VI.5	Print Procedure for Major and Minor Opcodes Called by the Generation Routine	86
VI.6	Storage Technique for Predefined Major Expressions and Associated Text	90
VI.7	Method Used for Storing Text of a Predefined Minor Opcode.	91

CHAPTER I

INTRODUCTION

The purpose of computer assisted instruction (CAI) is to complement the teacher in the dissemination of lesson material by creating an environment that parallels that of direct student-teacher interaction. This ideal is still far from being achieved, but CAI has made considerable headway toward a realization of this goal. However, with each advancement in the refinement of CAI, a corresponding problem has usually arisen. One such problem has been the need for simple but powerful author languages for writing lessons. Many system developers have been able to solve the problem; others have not. IBM's COURSE-WRITER III system is an example of one that has not. Though it has a simple author language, it is one that requires large amounts of time and effort to use effectively. The topic of this thesis is the discussion of a programmed system that presents a solution to the problems of time and effort in the writing of lessons in COURSEWRITER III.

The COURSEWRITER III language was designed to be modular and easy to learn at the expense of its own versatility. Constructed for the System/360 and System/370, COURSEWRITER is partially constrained by its accompanying hardware. Mostly, though, it is restricted by its own simplistic design. The restrictions imposed by the language constitute a great expense of time to the lesson author who must program the lesson. Therefore, there are a variety of lessons which

are not particularly compatible with COURSEWRITER. The system discussed later will allow implementation of some of the lessons and will save the author time, at the expense of the computer's resources, in preparing programmed instruction using COURSEWRITER III.

The main savings will occur at the level of the enumeration of many correct, wrong, and unanticipated responses. Currently, if the author wishes to specify several correct answers, for example, which are all of the same basic format, he/she must list each one with all of the resulting actions that should be taken should the student respond with a particular answer. This is considerably wasteful of time, boring to the lesson author, and a generally inefficient use of the teacher's educational talents.

Originally, the intent was to design this system to operate interactively as an interface between the user and the COURSEWRITER system. Two main problems exist that force the current version to be an offline implementation.

First, COURSEWRITER III is a copyrighted system that is leased to the University of Tennessee Computing Center (UTCC). In order to make modifications within the system, complete system software documentation would be required. This is not readily forthcoming from IBM.

The second problem is that COURSEWRITER III is a stand-alone CAI system that allows no external program intervention other than the 360 or 370 operating system. This fact diminished hopes of an online interface or even an insertion of datasets by a batch processed job.

The offline system does perform adequately and achieves its basic goals of reducing programming time and effort. It further diversifies the COURSEWRITER III programming language so that it may be used for lesson instruction that formerly could not have been done conveniently.

Finally, it is not the intention of this thesis to be a discourse on lesson writing or content or even a demonstration of a "perfect" CAI system (which COURSEWRITER III can never be). It is a discussion of the development and implementation of a programming system designed by the author to incorporate positive features of other available CAI languages.

CHAPTER II

CAI BACKGROUND

Individualized instruction has been the goal of educational strategists since formalized teaching began. It has long been assumed that as the student-teacher ratio approaches one, the learning time, retention time, and general motivational factors all reach an equilibrium that maximizes the absorption of material by the student. Over the years, however, particularly with the introduction of mass education, there have been several deviations from the goal of individualized instruction. As the number of students per teacher increases, teachers and students alike are frustrated and dissatisfied with educational processes and the quality of education itself.

Computer assisted instruction (CAI) marks a significant step toward returning to a one to one relationship between teacher and student. This is not to say that CAI is the solution to the problems in teaching today, or that CAI should replace traditionally administered instruction (TAI) for that is a valuation based on no concrete evidence. Current trends, however, seem to indicate that CAI has a future as an integral portion of the classroom and may be part of the solution to many of the difficulties in the teaching system.

A. History

CAI's history is rather brief since computers could not support an advanced processing network that CAI demands until fairly recently. There are developments that led to computer assisted instruction, nonetheless, that go back many centuries. The introduction of the textbook

in 1596 was the first attempt at standardization of course material and facilitated mass education. One of the necessities of effective teaching is strict course organization and content. Before printed material was available, the content of a course was based upon what the teacher could remember to say to the class. This method of teaching led to wide variances in amount and quality of material. The textbook had the advantage of stating the same material in a uniform way to everyone. This advantage was also one of its disadvantages. Not everyone learns at the same rate or in the same way and the textbook forced the student to adapt to its format. As a result, many books concerning basically the same topic were produced. This did not lessen the teacher's task of dealing with each student individually since the teacher could not select a different book for each student. Therefore, the problem remained that the students and teachers had to adapt to the style of the book chosen.

Despite its drawbacks, the textbook continued in extensive use for approximately 300 years with few noticeable advancements in teaching aids until 1927. Sidney L. Pressey, an innovative psychologist, was working with mechanized teaching. The machine he devised used the drill and practice style of teaching. The operation of the machine was fairly simple. Multiple choice questions were presented to the student who pushed one of four keys indicating the answer. The apparatus was constructed such that the entire material was presented twice. Then, the machine would again present only those questions that the students answered incorrectly until the student could answer each of those correctly twice. Pressey's (1932) premise for missed question repetition

was "as learning time progresses, the subject's attention is concentrated more and more on those items which are most difficult for him (p. 43)." According to Pressey, there were extensive possibilities and applications for his machine. Unfortunately, Pressey's research ended after eight years of work in 1932 because he was frustrated by the lack of acceptance and recognition of his contributions. He concluded his progress report with "I hope enough has been done to stimulate other workers (p. 43)."

Very few people must have been stimulated or impressed by Pressey's work because it was not until 1954 when a well-known psychologist in behavioral sciences B. F. Skinner released his experimentation with teaching machines. Skinner (1954) was interested in teaching students using immediate reinforcement for correct answers to given questions such that "aversive consequences of being wrong are reduced to a minimum (p. 54)." The student was sequenced through drill type questions on basic arithmetic by a machine that presented a question which was answered by moving appropriate key slides. After the student entered his/her answer, he/she would turn a knob that would ring a bell signifying a correct answer. If the student entered an incorrect answer, the knob would not turn and the learner was forced to try again. This process continued until the answer was correct, resulting in immediate reinforcement. With constant and immediate reinforcement, learning time was reduced. Skinner further supported his adaptation of the teaching machine in that it alleviated the drudgeries for the teacher of answering every question with "yes, that is correct" or "no, that is incorrect."

Norman Crowder (1960), a few years later, carried Skinner's work a few steps further with the development of a teaching machine which used a technique he called "intrinsic programming." The basis of intrinsic programming was that since there existed no infallible process to predictably achieve some end result, "one must have a means of determining whether the result has been achieved and of taking appropriate action on the basis of that determination (p. 87)." He accomplished this task by creating a system called Tutor which was a network that branched according to responses to questions given over small amounts of material. The student was given a paragraph to read which was followed by one question designed to check the student's comprehension. If the student responded correctly, he/she continued with the succeeding paragraph and question. An incorrect answer would branch the student to a remedial section, a restatement of the paragraph, or further explanation of the question. The question would, then, be restated. Hopefully, the student would respond correctly and continue.

The original medium for this system was a book but it was later converted to a machine which had the capability to show moving pictures and other visual aids. The text was stored on microfilm and was projected on a screen at the front of the machine. Though highly mechanical in design, this machine was one of the first to incorporate electricity.

Still, with all the developments of teaching machines, the institution of education was not shifting its methodology from TAI to mechanized instruction, though interest was growing. This was because,

as Skinner (1954) said, educators "accept the traditional as the inevitable (p. 87)." Few in education regarded the teaching machines as much more than experimental and, as a result, that was primarily the status they maintained.

The 1950's introduced the concept of programmed instruction with various texts and materials. But it was in the late 1950's that this concept began to find a place in education. This was because computers had been developed to the point that it was possible to create an environment of one to one relationships between student and teacher, except the teacher was the computer. People realized that the computer could be ideal for educational applications because of its capabilities for storage and retrieval of large amounts of information, its ability to branch several directions based on many criteria, and its ability to perform several tasks repeatedly that humans would find unattractive. With the introduction of the computer in education, the controversy began over whether CAI could ever be as effective as TAI.

The proponents and developers of CAI systems proposed that just as the computer was revolutionizing other factions of society, so would it revolutionize education. Several projects were accordingly undertaken by computer manufacturers, systems developers, and universities. Among these early developers was IBM. At the Thomas J. Watson Research Center in 1958, a system 650 computer was used to teach binary arithmetic. Also, at this time, System Development Corporation conducted a research effort examining the use of a computer to control varied displays on a variety of media. The objective was flexibility in branching using programmed instruction which necessitated considerable research in random access storage.

At Bolt, Beranek, and Newman, J. C. R. Licklider and John Swets developed a CAI system that, among other things, formed graphs in response to students questions concerning analytic geometry (Zinn, 1968a). These three projects were probably the first notable experimentation with computers in education, and even with fairly inconclusive results, CAI gained momentum quickly. By 1965, there were approximately 20 to 30 CAI projects with close to 50 unique curriculum packages (Zinn, 1968a).

Through the 1960's and 70's the use of computer assisted instruction has increased notably. It has not, however, increased as quickly as initially projected due to several factors. The three most important limitations have been cost, computer hardware, and computer software. For example, the implementation costs of the IBM 1500 Instructional Series were so exhorbitant, that the project was forced to remain as an experiment and only eleven systems were actually operational (Johnson, 1971). It has been because of these problems, though, that vast amounts of research have been conducted to improve the "industrial revolution in education" as Pressey termed it, giving rise to such systems as the TIES Project (1967), TICCIT (1973), and PLATO IV (1971).

B. CAI Modes

Computer assisted instruction is usually classified as one of four types: drill, tutorial, dialogue, or simulation (Frye, 1968). The drill method of CAI is probably the most common and easiest lesson to construct. In the drill mode, repetition of one class of problems is emphasized. Although very basic in design, a drill, or drill and practice as it is more commonly called, CAI course has its advantages. Primarily, the advantage is that course material is easily adjusted to the student's

competency level. Individualized instruction is maintained and the student can take a lesson until he/she masters the material. A well known example of the usage of drill and practice CAI is the system currently in use at the public schools in Palo Alto, California. The authors, Patrick Suppes and Richard Atkinson, have achieved excellent success with their programs to teach elementary arithmetic and language skills (Suppes, 1966).

The second type of CAI is the tutorial mode. A tutorial CAI course is characterized by a significantly more individualized approach to lecture, text, and examination material. Many hours of CAI material are available today on a wide variety of subjects. This is probably the result of the tutorial mode being relatively easier for an educational technologist to devise. The lessons usually guide the student through a variety of topics with the control of conversation still within the hands of the instructor. This type of lesson presentation does take more time to construct, though, than drill and practice. The Pennsylvania State University Computer Assisted Instruction Laboratory has undertaken many projects using tutorial CAI courses varying from mathematical to communications skills. Many of these projects have now become available course segments used as part of the curriculum for students at PSU (Berger, 1967).

The most advanced form, and unfortunately the most difficult to implement, is the dialogue mode. Here the student can interact directly with the controlling program by asking questions and responding in a reasonably free form. Many considerations must be taken by the lesson author using dialogue. First, he/she must evaluate the answer to a

question in terms of the forms the answer may take. Also, the author must consider past conversation in order to determine the scope of the programmed answer. In addition to these, the author must be concerned with the directions the next question will take and will that next question be within the realm of information that is being presented. Not many CAI systems are currently available that can even be programmed for such a task. The system developed at Bolt, Beranek, and Newman was such that it could handle dialogue. Another current system that was designed for this type of interaction is the PLATO IV system at the University of Illinois, a system which will be discussed in detail later.

A fourth type of CAI, simulation, is now emerging as an important mode as more advances in computer hardware and software are made. When performing simulation, the computer asserts the initial conditions about the problem. Following this, the student is then asked what action he/she would like to take to solve the problem. The computer evaluates the response and presents the student with the results and new conditions that have arisen. This process is continued until the desired result is achieved and the student can then evaluate the sequence of steps taken to arrive at that result. Extensive use of this type of CAI is also in effect on the PLATO system. This type of instruction is particularly useful as a replacement for laboratory experiments, especially ones where the equipment costs or inherent danger are prohibitive.

C. CAI Languages

There are several programming languages with which a lesson author can implement a CAI program whether or not it be one of the four types mentioned previously. Some languages, though, are more amenable to implementation than others. Charles H. Frye (1968) has classified author languages into four types: 1) conventional compiler languages, 2) adapted conventional compiler languages, 3) interactive computing and display languages, and 4) specially devised instructional author languages.

The class 1 languages are programming languages such as FORTRAN, PL/I, SNOBOL and COBOL. Although these languages have been used on occasion as author languages for CAI systems, they are generally too complex for the average lesson author to successfully implement a CAI lesson. Usually, if one of these languages is going to be used, systems programmers are employed to make necessary programming adjustments. An example of a system in wide usage today written in one of the class 1 languages is the Total Informational for Educational Systems, or TIES, project. The entire system which reaches in excess of 230,000 students in twenty Minnesota school districts was written and being updated in COBOL (TIES, 1967).

The adapted conventional compiler languages (class 2) were designed to alleviate many of the problems that arose when trying to use the class 1 languages as author languages. Revised FORTRAN was the basis for CATO and FOIL, TSA was extended ALGOL and MENTOR was an extension of LISP. Certain additions allowed by these languages were facilities for answer matching, record keeping, and sign on-sign off procedures for

the student. Even with the revisions, though, programming experience by the author was nearly mandatory.

The third class of programming languages is the interactive computing and display. This includes APL, BASIC, QUIKTRAN, and TELCOMP. The creation of these languages was an important step in changing the need for programming experience to enter a lesson. The characteristics of a class 3 language are primarily that they are easier to learn and have a conversational ability. They include the capabilities for answer matching, record keeping, and branching, but not at a sufficient level to support advanced CAI functions. They do provide a great deal of computational power and are oriented toward problem solving.

The fourth class of programming languages is the specially devised author languages. Examples of some current languages available are COURSEWRITER III, TUTOR, PILOT and PLANIT. There are a variety of these languages commercially available that are ideal for the implementation of a computer assisted instruction lesson. Since these are specially designed languages, they have the capability for ease of acceptance of answers, audio-visual display options, branching, and various record keeping facilities. Certain languages permit programming to match mispelled words and variations of words. It is usually possible to accept multiple forms of answers, for example, "4" and "four". When programming a lesson in a class 4 language, the lesson can be entered in a sequential and logical form that is meaningful to the lesson author. The syntax of these languages is usually fairly easy to learn and complicated programming routines can be reasonably programmed without a great deal of knowledge of the actual workings of the machine.

Also, the construction of the language is usually of a modular nature such that the author need not know the entire syntax of the language to adequately perform simple tasks. An example of this type of language is the COURSEWRITER III language for the IBM 360 to be discussed later.

D. Future of CAI

The directions that CAI may take are varied. It can move to a totally new means of education or it is equally possible for it to fail completely. CAI can and has been shown to be a worthwhile investment in many applications. However with all the information available about CAI, where it is implemented, in what forms, to what degree, and costs of it all, there is seldom any objective viewpoint taken concerning the overall effectiveness of the CAI systems (or for that matter any computer based instructional system). There have been numerous individuals and corporations willing to expound their observations and evaluations in as factual form as possible. These sources range from computer manufacturers to the students themselves. The basic unanswered question that arises, though, is "Does CAI have a future in the classroom based on what is known about it today?"

For the past ten years, educational enthusiasts and technologists have been claiming that CAI will totally revolutionize the school system and methods within the classroom. In many cases, they have been correct, and in quite a few, they have been wrong also.

Concerning the IBM 1500 Instructional Series, IBM determined that the evaluative ability of their machine was somewhat inconclusive, although they did find that learning time decreased and retention time of the material increased. Their main problem was "the controversy

what forms CAI should take and even the basics as to whether computer assisted instruction is a good thing at all (Johnson, 1971, p. 17)." It should be assumed by now that CAI is a "good thing" if for no other reason than the possibilities it has for diversifying the classroom environment.

The French conducted studies of CAI in 1971, and researchers concluded that "students found the task dry and couldn't see the point (Devos, 1970, p. 48)." This result was probably because the type of lessons were for the most part drill and practice which does become boring to a student after awhile. Note that while it is true that the learning process will be slowed if the material is presented in such a way that the student is totally disinterested, it should not be the basis of CAI to be an entertainment mode. The use of innovative presentation could be used to offset what eventually becomes boring to the student.

What many students find fault with in CAI systems is the formulation of the "drill monster" condition. In this type of learning program, the stress is placed on repeated exercises in a particular topic. Although this type of teaching rapidly disinterests the student, findings have shown that its benefits can outweigh the disadvantages. The results of studies by John F. Visonhaler and Ronald K. Bass (1972) show that where standardized achievement tests were used as a criterion for educational performance "generally CAI groups show performance gains of one to eight months over groups receiving traditional instruction (p. 31)." A reduction in learning time of this magnitude is hardly an advantage to be taken lightly, but a combination of tutorial, dialogue, and drill would probably enhance the results.

Even with performance studies that show impressive gains by the students, there is one problem that still remains: CAI is taking on a drill and practice format in many if not most applications. Along this line, James Koerner (1973) comments generally on this form of CAI lesson construction. "There is the problem of the programming itself--the 'software'--which has lagged far behind the hardware, imperfect as the latter has been (p. 43)." He adds that a brief sampling of the materials available show that the state-of-the-art is very rudimentary and "a long road lies ahead before a true technology can be achieved and developed (p. 43)." The solution to this problem is to invest more time in the programming of the material to be presented. This is limited by the availability of funds, time, and the technological media over which the course is carried.

If funding were available, time is still frequently a problem. From the author's experience with developing a CAI module (using COURSEWRITER III) the time involved to accomplish even the simplest task is enormous. Zinn (1968a) conservatively estimates that the time required for one hour of lesson material is 150 hours of development and programming. This arises from the need to integrate the material in a logical fashion and to account for the student's correct and incorrect answers. There is a need to reinforce correct answers appropriately and also explain to the student clearly why he/she was incorrect so that the student may improve. Still, these are problems which may be overcome with the passing of time. With continued research and development, it is very likely that hardware limitations and time limitations may be reduced to a minimum allowing for a less complicated form of CAI.

The future holds many possibilities for CAI. Presently, most educators agree that CAI cannot be effective in the traditional classroom for reasons such as cost factors. When the costs of CAI do become competitive with TAI, the simple solution would be to change the conventional classroom to fit the CAI type of teaching environment. Currently, however, CAI can be used optimally in the areas of high cost education. These are situations where students are paid salaries while learning, or areas where special student needs require low student-teacher ratios, as in the tutoring of children with learning disabilities (Butman, 1973). In these applications, CAI has been used effectively but these situations are not a traditional classroom environment. This is a beginning for CAI and when its success has been proven, it is likely that extended usage will evolve so that it is acceptable to an altered classroom.

CAI is definitely in its infancy and few predictions can be made about what directions it will take or even if it will even be as successful as its developers claim. It has only been recently that researchers have even begun to examine the long and short range effects of CAI systems. An interesting observation about CAI and its future was made by James Koerner (1973).

> Asking in 1973 about the future of educational technology is similar to asking in 1903 about the future of the aeroplane. How many could have envisioned an evolutionary line of flying machines that would produce the 747? Although it won't take another 70 years to produce an educational 747, it will take more than a few (p. 30).

There are many reasons to continue developing CAI and to try to reduce its costs. Although the deterrents show no signs of immediately vanishing, it is possible that within the near future, computer assisted

instruction will be amenable to almost all forms of education and the present observations about its effectiveness will be superceded by an advancement in the intellectual level of tomorrow's students.

CHAPTER III

CAI SYSTEMS

There are a variety of CAI systems in existence today in a variety of applications. Some of these systems are nothing more than a typewriter connected to a central processor while others are far more exotic in both input/output media and processing capabilities. This chapter will examine in some detail four systems currently available.

It is generally accepted by those considered to be authorities on CAI that every CAI system should have certain basic qualities and considerations in order for the system to be useful. These requirements can be classified into eight categories.

1. The programming language should be author directed with user orientation, lesson handling, record handling, conditional branching, and calculation provisions (Frye, 1968).

2. Answer match routines should be powerful with capabilities for exact match, keyword match, selected character match, percentage match, phonetic match, algebraic match, numeric match (within limits), calculated numeric match, and spelling algorithms (Frye, 1968).

3. Input/output devices should be versatile and student oriented.

4. The system should have capabilities for graphics and audio-visual aids.

5. Response time should be seemingly instantaneous.

6. The system should be able to retain extensive data for each student.

7. The hardware should be of utmost reliability.

8. Economic costs should not be so high as to offset benefits achieved by individualized instruction.

Some educational technologists may argue that the effective CAI system encompasses far more than the eight classes listed above and perhaps it does. However, the intent of this chapter is not to define exactly what the "perfect" CAI system is, for that is a highly debatable subject. The above classes simply serve as a basis for comparing and evaluating the systems to be considered.

A. PLATO System

The PLATO (Programmed Logic for Automatic Teaching Operation) system is probably the most well known and widely acclaimed CAI system available (Lyman, 1975). It is definitely the most sophisticated and approaches the concept of the ideal CAI system. The system grew out of many developments in the early 1960's and went through many changes before becoming the CAI system that it is today. In its first seven years, PLATO evolved from a single terminal connected to a digit computer to a system supporting 71 terminals, twenty of which were operable at a single time. The computer also changed four times beginning with an ILIAC I, replaced by a Control Data Corporation 1604, which was in turn replaced by a CDC 6000. Finally, PLATO was shifted to a CDC CYBER 73 (Lyman, 1975) in 1973 where it continues to serve as the central processor at this time.

The programming language that is used for PLATO is called TUTOR. This is a highly author directed and extremely versatile language. The current version is an interpreted system that has several commands that

are very readable even to an author with little programming experience. With simple commands such as WRITE and DRAW, a lesson author can construct very interesting and pictorial lessons with little difficulty. When drawing figures, the author need only specify end point locations on the display panel and the system will automatically draw connecting lines.

Although TUTOR is a simple language to learn, it is very powerful. Features for computation are advanced such that calculation constructions closely parallel natural algebra with implied multiplication and superscripts as exponents. All calculation sequences are compiled into CDC machine code to maximize execution speed (CERL, 1974). Therefore, the student can enter an algebraic answer and the system will evaluate it and compare the result to predicted answers.

Extensive graphics capabilities may be employed by using commands for drawing circles, arcs, or variable sized characters. These commands are simple in format; they only require a location on the display screen as their text. All drawings are then produced by internal system routines.

The answer judging capabilities of PLATO are especially impressive. With most other systems, many anticipated responses must be specified to handle phonetics, spelling errors, and incorrect words. In contrast, PLATO has efficient algorithmic answer matching routines that allow the system to carry on dialogues with students or judge the response with emphasis on spelling, vocabulary, or even entire concepts. Performing this task requires very little effort from the lesson author.

For example, if the author writes

ANSWER <it,is,a>(right,rt)triangle

several anticipated answers have been specified. The words in brackets indicate that these are unimportant words in the answer. Words in parentheses indicate synonyms, and the word "triangle" is the designated correct answer (Sherwood, 1971). Therefore, if the student enters

it is a tringle

the response will be judged as correct even with the spelling error. The answer match routines are so sophisticated that most of the work in recording predicted responses is performed automatically. If unnecessary words appear, they are crossed out and misspelled keywords are underlined, thus informing the student of the error.

Keyword analysis is also highly developed. Predicted responses may be programmed such that a question may be entered by the student and analyzed by the PLATO system to extract the content of the question. If undefined words appear, PLATO will apologetically list the undefined words and ask for a restatement of the question.

Authoring of a lesson is done at the terminal and the language permits rapid switch from author mode to student mode and vice versa. This allows the author to immediately test the course segment after it is entered. Corrections can be made easily by the text editing system available. It is not necessary to reenter entire lines of code to be corrected. PLATO is constructed to allow editing of specific portions of lines resulting in a very user oriented system.

The language also permits PLATO users to communicate between terminals. Thus, one author can consult another for programming advice or other assistance. This is also extended to allow an author at one terminal to monitor student activity at other terminals. The lesson author can watch each student's response and perhaps recognize areas of course presentation that need reworking and development.

Another important feature of the PLATO authoring system is the ease and thoroughness with which student response data can be collected. There are a variety of options available to the author that include automatic data collection of time spent at the terminal by each student, number of right and wrong answers given, detailed data, such as specific wrong answers given, and number of requests for help.

Responses for requests for help are specified by the author who programs the text presented to the student for each request, usually giving the student more information each time. The language is designed to allow unlimited responses to help and is under direct author control.

The plasma display panel at the student terminal is possibly what PLATO is best known for. Invented by University of Illinois staff, the small black screen that glows orange with text makes possible course presentation that could not adequately be accomplished on a conventional screen. Some of the more interesting features of this terminal are given below:

1. An 8½ inch square plasma display panel that is readable in a brightly lighted room without eyestrain.

2. Permanent storage on the display screen without flicker. Absolutely no refreshing of the display panel is required.
3. Self contained character and line generator.
4. A character writing speed of 180 characters per second and the capability of displaying up to 2048 characters on the screen.
5. A line drawing speed of 60 lines per second.
6. A character repertoire of 256 characters, 128 of which are alterable via the computer program.
7. The ability to transmit and receive data on voice grade telephone circuits.
8. A random-access slide projector for rear projection of static information on the display screen.
9. Additional input/output channels for the control of auxil equipment.
10. An optional random access audio response unit (Stifle, 1974).

This terminal was designed with both author and student in mind and was further developed to be compatible with typical communications links for remote placement. It contains no cathode ray tube display so that low grade transmission lines can be used. The projector in the rear of the terminal uses microfilm displays and allows computer generated images to be superimposed on the static displays which may even be in color.

The display screen itself was a giant step forward in CAI terminal design. It consists of a flat glass plasma panel with a 512 by 512 array of individually addressable points under computer control (CERL, 1974). This allows for detailed graphics to be presented and adds immeasurably to the overall course presentation. The developments of this terminal alone has probably resulted in more outside support of PLATO than any other single advancement. At the CDC press conference in April of 1976 demonstrating the PLATO system, most people were awestruck by the terminal's capacities.

> Reporters wandered for hours from terminal to terminal at the mid April affair watching demonstrations of courses on English, Russian, chemistry, genetics, accounting. Frogs jumped, flies multiplied, and checkers moved on incredibly flexible displays (Pentages, 1976, p. 187).

This terminal's capacities far exceed any other developed to this date. With continued research it will likely maintain its excellence; plans are in process for an even more advanced terminal.

The PLATO hardware involves a complicated system of main processors and peripheral processors. The system requires the services of a totally dedicated CYBER 73 computer that supports 950 terminals at various sites around the world (Lyman, 1975). The network also contains a two million word unit of extended core storage as the swapping facility for the student programs. A unit of this nature usually does not appear as part of most CAI hardware systems because of the excessive costs. PLATO's premise is that no extensive course swapping will be used by retaining several lessons all in main storage at one time. "It is overall performance that matters and increased memory costs are offset by the elimination of the high swapping overhead with drastic improvement in quality" (Bitzer, 1973, p. 9). In most other systems, course swapping is done between core and disk which results in considerable overhead in software and increased response time. PLATO's hardware configuration allows a mean response time of 0.2 second (C , 1974).

One of the major problems that PLATO has at this time is significant downtime of the system due to hardware and software problems. This is attributable to the fact that PLATO is still in a developmental phase and experimentation is performed constantly. Despite this fact, has announced that its version of PLATO is now a commercial product of that corporation (Pentages, 1976).

PLATO's other main problem is that it is expensive. Predictions were that when PLATO is fully implemented the cost would be less than one dollar per student per contact hour (CERL, 1974). Now, though, the costs are significantly higher. CDC estimates its version to be between $11 and $12 per contact hour (Pentages, 1976). The terminal alone costs at least $5000 (Stifle, 1974), and with further developments, the cost will surely rise.

PLATO does have a promising future and already has over 4500 completed lessons available (Lyman, 1975). The developers, such as Don Bitzer, foresee "millions of terminals" (Lyman, 1975) all within one network around the world. This is not only highly optimistic but very unrealistic. The current capacity of the system is only 950 terminals and requires a very large machine. The physical limitations of a computer to process data for millions of terminals leads one to doubt whether the goal will ever be achieved.

B. TICCIT System

The Time Shared Interactive Computer Controlled Information Television (TICCIT) (Volk, 1974) from MITRE Corporation represents a new philosophy of development for a CAI system. Begun in 1973, TICCIT emphasizes systems concepts in course preparation and delivery. It also emphasizes a commercial aspect of CAI in that its goal is a market success for MITRE's product.

The system concept of TICCIT involves a procedure similar to the processing of any product placed on the market for sale. According to MITRE, in the future, CAI software will be written by lesson authors, coded by programmers, and packaged for sale by marketers. The entire

concept is a

> . . . highly organized, highly formalized procedure for instructional program production and an extreme contrast to older teacher-author programming concepts (Adams, 1973, p. 24).

The developers feel that allowing teachers to also be programmers lessens the overall course quality. TICCIT programs are designed to be implemented in class 1 languages. Therefore, the lesson author would have to be totally familiar with the computer operating system to program the lesson efficiently.

The result of the authoring philosophy has given rise to a totally different type of course preparation. Though TICCIT has capabilities for online authoring, most of the courseware is done offline. Here the author fills out forms that state the text, questions, graphics, predicted responses, and actions taken for predicted responses. These are then entered into TICCIT by programmers and the lesson is compiled. Online and offline editing is permitted of the existing program code in case changes are necessary.

The answer matching routines are reasonably powerful and comparable to those of the PLATO system. Accounting for phonetic spellings can be programmed and calculations can be performed as necessary in order to evaluate the student's response. Lexical analysis can be performed as needed and numeric matches within specified limits are included as part of the routines. The system also has the capability for partial answer processing.

The author, as in PLATO, does not have to specify several spellings of words in order to achieve a match of a predicted response; spelling algorithms are a part of the answer matching routines.

Data concerning each student, logon time, response time, etc., are stored on magnetic tape for later analysis depending upon options specified by the lesson author. In addition, help requests are handled by a specific file in the operating system that is very useful. This file of programmed text can reroute the student through new material or over that which has already been programmed. It may also be used for restatement of questions and ideas, then returning to the original question.

The student terminal is dependent, as with most CAI systems, upon static displays on a CRT display screen. This means that terminals beyond 1000 feet from the processor site must be connected via cable television lines due to refresh requirements. This method of connection is more expensive than voice grade telephone lines such as those needed by PLATO.

The terminal itself is a color television with audio capabilities connected to a keyboard that may either present up to 17 lines of 43 characters per line with 512 characters defined at one time (characters are programmable) or present full color and sound movies stored on video tape. A switch is on the terminal for each type of mode.

Graphical capabilities exist with TICCIT that are developed off-line as frames. These are then scanned optically and stored digitally for projection on the screen. Random access frames and audio passages are also programmable. Unlike the PLATO terminal, this terminal is less versatile in functions that can be performed simultaneously. The keyset, though, is very student oriented with such function keys

are EXAMP (example request), HELP, SUPER HELP, and ADVICE. The operating system will automatically branch to the appropriate files depending on which keys are depressed.

The computer hardware is the most recent developed and consists of a Data General Nova 800 with 49K of storage linked with a terminal processor and several direct access devices. This configuration can serve up to 128 terminals. Each component used has been tested for reliability, and thus system downtime is claimed to be minimal. MITRE reports that there will be very few occurrences of system failure and that any occurrence will rarely exceed thirty minutes (Volk, 1974).

Also with most recent developments in hardware, response time has been estimated to be 0.8 seconds with 128 terminals in use. The higher response time results because of the swapping scheme utilized. Whereas PLATO uses in-core swapping (which is the fastest and most expensive), TICCIT uses core to disk swapping (which is less expensive but slower).

The TICCIT system was designed to be low cost but efficient. MITRE estimates the cost of a complete system with 128 terminals at $441,300 (Volk, 1974). Considering the cost of many other systems available, this is competitive and apportioned over 128 terminals results in $450 per terminal allocation due to the fact that most of the expense is in the processor and not the display station (Volk, 1974).

TICCIT is still in a developmental stage at Brigham Young University and evalutions have not yet been made. Planned commercial implementation is a few years away as many operation and system problems remain to be solved.

C. GIZMO System

The GIZMO System (Davis, 1973) was also a development at the University of Illinois at Urbana-Champaign which was begun in 1971. Designed to run on a Digital Equipment Corporation PDP-11 mini-computer, the philosophy behind its creation was that drill and practice lessons should be easily programmable and GIZMO would represent an extremely author directed programming system to perform this task. Several classes of problems were eliminated in order to limit burdens being placed upon the lesson author. Such problems were display mechanisms, handling of audio output, and dynamic supervision of programming segments which can be accomplished in PLATO and TICCIT.

Another goal of the system was a minimization of cost in both hardware and operation. The present version accomplishes this task by using a typewriter keyboard as the input/output medium connected to the PDP-11 central processor using the Educational Timesharing System (Oxley, 1972) also developed at the University of Illinois.

Certain problems are inherent with the limited amount of hardware used. The teletype is slow, noisy, and has very little capability for graphics. Therefore, the type of lesson that may be programmed is primarily constrained to drill and practice format. Also, the PDP-11 machine can only support eight to twelve terminals (Oxley, 1972). This is acceptable if the number of student users is very small. However, for a number this small, an individual instructor could be assigned that would be much more cost-effective than the CAI system. Increasing the number of students also creates a problem that not enough terminals would be available to adequately service the student.

The authoring language used in GIZMO is "ideally suited to writing a syntax recognizer" (Davis, 1973b, p.3) and is very simple in operation. All lesson requests are made by GIZMO so that the structure of each lesson is GIZMO-directed. This makes authoring easy as all that is required is to respond to GIZMO's requests.

Specifying predicted answers in GIZMO is done with a Backus-Naur like programming language. Using this format, several predicted answers can be specified in one statement. Text typed to the student is specified as numbers which are later defined when GIZMO requests their definition. The lesson once entered is then compiled in order to speed retrieval of information and minimize disk storage.

The answer matching capabilities are surprisingly advanced using this uncomplicated language although it does require an exact match for a response to be judged as correct. A backtracking algorithm is used to compare the predicted response with the student's response such that partial answer processing can be accomplished and various messages can be printed depending on successes or failures in the matching routines.

Evaluation of student responses is also supported by GIZMO. This can be optionally called by the author in the program. GIZMO's evaluation function performs the necessary calculations and attempts to match the results with specified results. Branches may be taken depending on a match or no match.

The student terminal is far less elegant than for either PLATO or TICCIT. However, it seems adequate for the type of lesson usually being presented. The system was designed to teach introductory assembly language which seldom requires audio-visual aids. Response time for

this terminal is considerable due to the amount of time required to print the question and associated messages.

The GIZMO system performs well its designated task and achieves the basic goals established by its author. The answer matching capabilities of this system gave rise to the development of a new programming system discussed in Chapter V.

D. COURSEWRITER III System

COURSEWRITER III is IBM's contribution to the CAI world. More a programming language than a system, COURSEWRITER does have the capabilities to be used as a viable CAI system. The IBM 1400 Instructional Series using COURSEWRITER I served as the basis for the present language. This was later revised to form COURSEWRITER II for the IBM 1500 (Zinn, 1968b). With the introduction of the System/360, COURSEWRITER III was designed to run as a teleprocessing system in the high priority partition of the central processor (IBM, 1973b).

COURSEWRITER, without the hardware originally intended for use, is not an impressive system at all. Student terminal stations consisting of the IBM 3270 CRT displays were the original terminals to be used. Most academic institutions found these to be far too expensive to support a CAI laboratory, and, as a result, currently use 2741 typewriter or lesser quality input/output stations.

The 3270 is a highly advanced CRT display screen when run with COURSEWRITER, that allows limited graphics, emphasis of keywords (appear brighter on the screen), forward and reverse paging in the course, and immediate or delayed hardcopy of the student's sign on record.

Audio-visual devices are also supported by this terminal which allows random-access under computer control. Displays are on microfilm display devices and tapes contain audio passages. The terminal also supports light pen entries.

COURSEWRITER III is one of the most common, if not the most common, CAI systems in existence. The primary reason is cost in terms of machine dedication, operation, and actual rental fees. The minimum system configuration for a System/360 with OS is 55K (IBM, 1973a). For most 360's, this is a very small amount of space and allows the machine to be used for other processing besides the CAI system tasks. The operation requires no computer operator intervention as the system performs its own management except for startup and termination. The annual cost for COURSEWRITER is somewhere near $3300 and the cost of each 2741 is approximately $1320 annually.

The author language used on the system is also called COURSEWRITER III and is extremely simple. The language consists of interpretable code that is a fixed-field three character operation code followed by text. It is because this language is very author oriented that advanced answer matching is difficult. The language provides no algorithmic matching or phonetic capabilities. Therefore, if these are needed by the author, he/she must enumerate explicitly the many possible responses and replies which is a very tiresome and frustrating task. Also, without the 3270 terminal, editing the material must be done by retyping complete lines over. A more detailed discussion of the language follows in Chapter IV.

The language does have extensive record keeping facilities available to the author. The data may be printed, stored on tape, or stored on magnetic disk for later evaluation. In certain modes of operation, COURSEWRITER keeps many of the important records automatically.

Comparison of systems such as PLATO and TICCIT with COURSEWRITER is difficult since both PLATO and TICCIT are far more advanced than COURSEWRITER. In certain aspects the GIZMO system also exceeds the COURSEWRITER capabilities. However, COURSEWRITER can and is being used as a successful and effective means of education at several levels of instruction. For example, Ohio State University uses the COURSEWRITER system for teaching supplements to many Home Economics and medical training courses. It is very rare, though, when COURSEWRITER lessons are the sole means of instruction as the PLATO lessons in Illinois are.

CHAPTER IV

COURSEWRITER III

COURSEWRITER III is a CAI programming language integrated with an interpretive system that may be used to implement a drill and practice, tutorial, or dialogue type of lesson. The operation of the system is relatively simple. A scanner accesses each statement in the order that it appears and identifies the operation code (opcode) for that statement. Once the identification is complete, appropriate action is taken by the system. The COURSEWRITER program is not compiled, though it is modified somewhat internally for compatibility with output devices.

Because the language is author directed, the syntax is fairly trivial, particularly for someone familiar with programming computers. The language is not constrained by any specific type of lesson content, and the author may use almost any technique of programmed instruction that he/she chooses. The author also directs the computer in both programming and lesson modes.

The following discussion is a general overview of the basics of the language. It is not designed to be an indepth description of the entire COURSEWRITER III programming language. For a complete delineation of the COURSEWRITER III language, see the IBM COURSEWRITER III Version 3 Student Text. This overview is necessary in order to gain a more thorough understanding of the extensions of COURSEWRITER discussed in Chapter V.

The COURSEWRITER III programming code is a series of operation codes followed by a blank and usually some sort of text. Each of these

opcodes consists of a two letter abbreviation for its function. For example, "QU" means question. These opcodes are classified into three categories: 1) problem, 2) major, and 3) minor. Table I shows a listing of the three types of opcodes and their associated meanings.

The problem opcodes control the flow of the course as it is taken by the student. For example, the problem opcode QU heads an answer group that contains major opcodes. The text associated with a QU is the question presented to the student. When all the text has been typed, the system waits for a response from the student. The RD (read) opcode is similar to the QU except that it does not compare the student response to anything. It is used for presenting material and allowing the student to grasp what is shown before continuing. A response from the student (usually pressing RETURN) allows processing to continue. The third problem opcode, the PR, is used with no text and may be invisible to the student since it also does not post a wait to the system. It is frequently used to head groups of minor opcodes that reset counters or modify system parameters during course execution. Messages may be typed to the students, but no answer matching occurs in a PR group. Some format examples of the problem opcodes are:

QU WHAT IS THE SUM OF 5 + 5?

RD READ CHAPTER 2 AND THEN PRESS RETURN TO CONTINUE

PR (no text)

Another group of opcodes are the major opcodes. The CA (correct answer), WA (wrong answer), and UN (unanticipated) are the most commonly used. It is within these statements that the actual answer matching occurs, and they must follow a problem opcode. A statement that

TABLE I

COURSEWRITER OPCODES AND
ASSOCIATED MEANINGS

Problem	Meaning	Minor	Meaning
QU	question	TY	type
RD	read	AD	add
PR	problem	SB	subtract
		MP	multiply
Major	Meaning	DV	divide
		EP	enter and proceed
CA	correct answer	LD	load
WA	wrong answer	BR	branch
AA	anticipated answer	ED	edit
UN	unanticipated answer	FN	function
CB	alternate correct answer	AU	audio position
WB	alternate wrong answer	FP	film position
AB	alternate anticipated answer	PA	pause
NX	no match	DE	display erase
EA	end answer		

Source: IBM Corporation, COURSEWRITER III Version 3 Author's Guide, White Plains, New York, p. 7, 1973.

has a CA opcode contains a correct answer to the given question. When the student's response matches the author's correct answer, any associated minor opcodes are executed and the system skips to the next problem opcode in the sequence ignoring any intervening statements. The WA is very similar to the CA in that when its text field is matched, any associated minor opcodes are executed. However, after this, the system does not skip to the next problem opcode, but instead permits the student to try again. Since the author will want to anticipate wrong answers, he/she may use the WA opcode to do so. The UN opcode is used at the end of an answer group to account for all unanticipated answers. The author cannot possibly account for every response from the student, therefore, using the UN opcode, any response from the student that does not match the CA or WA opcodes will result in the text of the UN being printed and the student is given another try. To avoid hanging the student in an infinite loop, UN's may be stacked such that the first UN is executed the first time that an unanticipated answer is given, the second the second time, and so on. The text of these UN's could be hints for obtaining the correct answer. The last UN might be the answer itself with a request for the student to enter that information so that the lesson may continue.

The AA (anticipated answer) opcode is usually used in partial answer processing. As with the CA and WA opcodes, a match of the text field results in the execution of all minor opcodes. The main difference, though, is that the action taken by the system depends on the next major or problem opcode. For example, if a UN appears, the student will be given another try at the correct answer. If one of the other major

opcodes or a problem opcode follows the AA, the action taken by the system corresponds to the type of that particular opcode.

Three less frequently used major opcodes are the CB, WB, and AB. Similar in nature, they provide alternates for the CA, WA, and AA, respectively. An alternate answer may only be used if it accompanies an anticipated answer. For example, a CB may only be used if there is already a CA that directly precedes it. The alternate answers are usually used as a way of comparing the student's response to different forms of the same answer. The following sequence of COURSEWRITER code exemplifies the use of the CB opcode:

```
QU HOW MANY X'S APPEAR HERE?  XXXXXX
CA 6
CB SIX
CB VI
CB 110 (binary)
```

Four separate CA's could have been used, but if there had been minor opcodes specified in this example, the same sequence of instructions would have to have been duplicated four times. Using the CB opcode, minor opcodes would only appear once. The construction and rules are the same for the WB and AB.

It should be noted here that in the case of each of the major opcodes, except the UN, an exact match is required before appropriate action can be taken. However, periodically the author wishes to accept any answer that is essentially correct. For instance, "CARD REEDER" might be accepted for "CARD READER." Used with the line modifier, which is a "(L)" appended to the opcode, the "*" and "&" characters

may be employed for this purpose. The two symbols may be used together and allow for scanning of the student response to obtain a general idea of the nature of the student's answer or question.

Inserting an asterisk in the predicted answer, such as "CARD RE*DER" allows a match no matter what character the student places after the first "E". Thus, "CARD REDDER", "CARD REEDER", and "CARD RE DER" are all valid entries and would result in a match.

The ampersand is used for multiple character elimination. Its placement in the answer causes all intervening characters to be deleted before answer matching is attempted. If the answer appeared as "&CARD READER&", "OLD CARD READER", "I BOUGHT A CARD READER YESTERDAY", and "PLACCARD READERS" would be recorded as matches.

The NX (no match) opcode is used when the student's response did not match any of the text fields of the preceding CA, WA, AA, CB, WB, or AB opcodes. The command contains no text and, when it is encountered, all minor opcodes up to the next problem opcode are executed. It can be used for branching, restting counters, setting switches, or altering course parameters only on condition of no match by the student.

The last major opcode, also used without text, is the EA (end answer) opcode. When the EA is encountered, all following minor opcodes up to the next problem opcode are executed if and only if there was a branch taken by the program to a label directly preceding the EA. If the system encounters the EA while processing sequentially, all minor opcodes are skipped and execution continues with the next problem opcode.

There are several minor opcodes that may be used to type messages to the students, perform arithmetic calculations, or present audio/visual aids. Minor opcodes are always preceded by problem or major opcodes and are only executed when one of the two types is executed. Therefore, minor opcides that follow a WA will not be executed if a match occurred with a CA.

The TY (type) opcode is probably the most frequently used minor opcode. It is used for typing information to the student. The information that is sent is exactly what appears in the text field of the TY statement. Quite frequently it is used to congratulate the student for a correct response or to advise the student for an incorrect answer before moving to the next problem sequence.

The arithmetic functions are accomplished using the AD (add), SB (subtract), MP (multiply), and DV (divide) opcodes. All calculations are performed using counters. In COURSEWRITER III there are 31 counters denoted by C0 through 31 with a decimal capacity in the range of -32,767 to +32,767. The form of arithmetic operations is

opcode CM or decimal constant/Cn

where $0 \leq m \leq 31$ and $0 \leq n \leq 31$.

Examples of possible forms are:

AD 1/C2	add 1 to counter 2
SB C5/C6	subtract counter 5 from counter 6
MP 10/C0	multiply counter 0 by 10
DV 5/C21	divide counter 21 by 5

The results of any operation are always stored in the second argument, and all calculations are constrained to the use of integer values.

The information that a student enters is contained in a 100 character buffer that can be readily accessed by the author. In addition to this buffer, designated B0, there are five more buffers, identified as B1 through B5, each with a 100 character capacity. Information or text may be stored in each of the buffers and referenced later by its number. For example, if the message "VERY GOOD, YOU ARE CORRECT" is used repeatedly throughout a lesson, this text could be stored in a buffer, such as B3, and typed to the student with the code

```
TY B3
```

Each time this statement is executed, "VERY GOOD, YOU ARE CORRECT" is sent to the student. Buffer designation may also be used with the major opcodes such as

```
WA B4
CA B1
AA B2
```

This reduces the amount of time in entering the program and the length of the program itself. Chapter V will describe additional methods.

Storing information in a buffer is accomplished by the LD (load) opcode. This opcode is also used for setting switches, parameters, and counters. The LD opcode has two arguments. Argument one is the value that is loaded and argument two is the destination for that value. The statement "VERY GOOD, YOU ARE CORRECT" can be stored in buffer 3 by the statement

```
LD VERY GOOD, YOU ARE CORRECT/B3
```

Counters may also be initialized using the same general format. For instance,

```
LD 0/C1
```

sets counter 1 to 0. The statement

```
LD 0/C0-30
```

initializes all counters to 0.

Occasionally, the lesson author wishes to obtain information from a student that cannot be anticipated directly, such as the student's name. The EP (enter and proceed) opcode permits halting of the execution stream to allow the student to enter a response. Often the EP opcode follows the question "WHAT IS YOUR NAME?" and precedes the LD opcode. This allows the author to access data at execution time and maintain that information throughout the sign on time of the student.

There are 32 switches (denoted Sn for $0 \leq n \leq 31$) each with the value of either 0 or 1 representing off or on respectively. The author can test the state of any of the individual switches and vary the course flow accordingly. Switches may also be cleared or set using the LD opcode. As an example, to set switch 24 to off, the statement

```
LD 0/S24
```

would be used. All switches can be turned on by the statement

```
LD 1/S0-31
```

Thirty-two parameters exist for the author's use both to vary course flow by providing additional criteria and to provide for alternate forms of processing by the COURSEWRITER system. Table II

TABLE II

PARAMETERS AND ACTION
TAKEN IF SET TO 1

Parameter Number	Action Taken on Student Responses
0	Change to lower case.
1	Remove space characters.
2	Remove line feed characters.
3	Remove backspace and the immediately preceding letter.
4	Remove tab characters.
5-7	Unassigned.
8	Requests for help are not allowed.
9	Go to requests are not allowed.
10	Send HELP NOT ALLOWED for all help requests.
11	Send GO TO NOT ALLOWED for all go to requests.
12	If no UN statement is present at the end of an answer group, operate as if such a statement were present with a text of INCORRECT.
13	Restart the student at the label in return register 5 (where he signed on).
14	Branch to the label in return register 4 when go to is used.
15	Poll for CA, WA, or AA opcodes following PR's.
16	Store student record at sign-on rather than at sign-off.
17	If a branch to an invalid label is requested, branch to the label in return register 3,
18	This bit on and in student mode causes the contents of counter 30 to be typed.
19	This bit on causes the normal carrier return following a student response to be omitted.
20	Inhibit student comments from being recorded.
21	This bit on causes counter 29 to contain the number of characters in the student response including the EOB.
22-29	Unassigned.
30	Inhibit execution of AU and FP opcode statements.
31	This bit on means a transmission error occurred on the last student response.

Source: IBM Corporation, COURSEWRITER III Version 3 Student Text, White Plains, New York, p. 38, 1971.

shows the various conditions by having each of the parameters on (set to 1).

The parameters have default values that the lesson author need not change unless it is necessary. Changing parameter values is also done with the LD opcode in much the same way as setting switches. The parameters are designated P0 through P31 and their values may be altered by LD commands, for example,

LD 0/P0

Groups of parameters may be changed by statements such as

LD 1/P10-15

Combinations of parameter values also establish conditions for operation as can be seen by Table III. With parameters 8 and 10 on, the student may not receive help from the system should that restriction be requested by the author. Other parameter groups create additional operational modes. The individual parameters and parameter groups eliminate the need for the author to check the student's response and alter it to a recognizable form. Therefore, they are of great importance and are used often.

Varying course flow either unconditionally or conditionally is accomplished with a BR (branch) opcode. The unconditional branch is simply

BR label-name

where the label-name is a one to six letter name established by the author as in any other type of programming language. This type of branch is useful for skipping around in the course in the case that

TABLE III

SYSTEM ACTION FOR OCCURRENCE
OF PARAMETER INTERACTION

For Parameters 8 and 10

Value	Action on Student Responses
0 0	Help is allowed.
0 1	Help is allowed.
1 0	Help is not allowed. If help is typed, treat it as a normal response.
1 1	Help is not allowed, and message HELP NOT ALLOWED is sent.

For Parameters 9, 11, and 14

Value	Action on Student Responses
0 x 0	Go to is allowed.
1 0 x	Go to is not allowed. If the student types GO TO, it is treated as a student answer.
1 1 x	Go to is not allowed. The message GO TO NOT ALLOWED is sent.
0 x 1	A go to is allowed using the label in return register 4.

(x = either 0 or 1)

Source: IBM Corporation, COURSEWRITER III Version 3 Student Text, White Plains, New York, p. 38, 1971.

the student has demonstrated capabilities in an area and should progress to more complicated material or has demonstrated deficiencies and must receive remedial instruction.

Conditional branching is accomplished by testing values of counters, states of parameters, states of switches, or any combination of the three. The format of the conditional branch is

BR label-name/condition(s)

As in most programming languages, there are relational operators in COURSEWRITER. These apply to comparison of contents of counters or comparison of contents of counters and constants. Their abbreviations and meanings are given below:

L	first counter less than second counter
E	first counter equal to second counter
G	first counter greater than second counter
LE	first counter less than or equal to second counter
GE	first counter greater than or equal to second counter
NE	first counter not equal to second counter (IBM, 1973b, p. 32).

The condition that is to be satisfied is constructed by placing one of the relational operators between the operands, separated by commas. Some examples are:

BR START/C5,G,C2	means branch to start if the contents of counter 5 are greater than the contents of counter 2.
BR TEST3/C2,L,C4	means branch to test 3 if the contents of counter 2 are less than the contents of counter 4.
BR GO/C9,E,15	means branch to go if the contents of counter 9 are equal to the constant 15 (IBM, 1973b, p. 32).

Branching may also be dependent upon states of switches or parameters. In this case, no relational operator is needed and the switch or parameter being tested is separated by a comma from the

desired state for the test. Thus,

```
BR START/S10,0
```

means branch to start if switch 10 is off. The statement

```
BR END/P25,1
```

causes a branch to END 1 if parameter 25 is set to 1. Conditions may be combined by separating each with a "/". The statement then causes a branch only if all are satisfied. To branch to the next problem opcode if C5 is 10, S6 is on, and P18 is 0, then

```
BR PR/C5,E,10/S6,0/P18,0
```

would be used. A 100 character limit is the only restriction on the number of conditions that may be tested in one statement.

Sometimes in the stream of the program the author needs to modify the students response by eliminating characters or shifting to all lower case letters. This is the purpose of the ED (edit) opcode. Its form as a minor opcode is ED and one or more of the following options:

SH	force to lowercase
SP	remove space characters
LF	remove line feed and return characters
BK	remove backspace and immediately preceding character
TB	remove tab characters
ALL	remove all of the above (SH, SP, LF, BK, and TB)
CHN	remove the character represented by n

Suppose the author designates the correct answer to a question with no spaces. Then if the student responds with spaces in the answer, there is no match even though the student may have been correct. To account for this problem, the author might program a sequence

similar to the following:

```
QU  WHAT IS 35 X 35?
EP
ED  SP/LF
CA  1225
```

Here the student may space as he/she wishes and a match will still occur. It is not necessary to repeat this sequence of editing with every question as some of the same tasks can be performed by parameter editing.

The FN (function) opcode is highly limited in application to usual lesson programming. The opcode calls an external function that is a core-resident portion of the COURSEWRITER system. It cannot be author defined as in many other programming languages. A function routine in COURSEWRITER III must be written in IBM 360 or 370 assembly language and added to the system. The general format is

```
FN function-name/parameters
```

Some functions existent on the University of Tennessee COURSEWRITER system are a random number generator (called RN), calculate function (CALC), text editing function (ED), limit function (LT), and mark character function (MC).

The remaining minor opcodes refer to the use of audio/visual devices and have no importance in this discussion since they are not supported by the system available at the University of Tennessee and are not related to research being described herein.

The COURSEWRITER language has many other features which attempt to enhance and simplify its use. These additions, which involve more complicated syntax, include statement modifiers, return registers, and restart points. All are extensions of the basic opcodes discussed previously. One highly useful addition, hitherto unmentioned, is the capability for indirect addressing. For example, a branch to START could be made depending on the state of the switch indexed by the contents of counter 4. The construction of this statement would look like

```
BR START/S(C24),0
```

Counters, switches, parameters, buffers, and return registers may all have indirect references which expands the capabilities of the system.

This chapter has described most of the features of the COURSE-WRITER III system. As was the intent of the developers, any beginning programmer should be able to learn to use the language quickly. Of course, the simplicity of the language can also cause problems for the programmer. Lessons are often extremely lengthy and require large amounts of time to construct and program. It is this class of problems that is solved by the development of the system discussed in the next chapter.

CHAPTER V

SLATE: AN EXTENDED COURSEWRITER III

Programming in COURSEWRITER III is by no means a complicated task as long as the lesson being presented is a drill and practice with very few predicted responses. However, more innovative course presentation or adding more predicted responses can become very involved and may require an extensive amount of time to develop and program. Since the lesson author should be able to concentrate most of his/her time on course content and not on programming, a system was designed to accomplish two tasks: 1) relieve the author of routine and boring programming, and 2) extend the capabilities of the COURSEWRITER III programming language itself. The SLATE (Symbolic Language And Text Expander) system is a preprocessor for COURSEWRITER III code that allows a lesson author to 1) limit the predicted responses by the student to a class of answers, 2) define several sequences of characters by single symbols for later reference, and 3) define several lines of code by a single symbol also for later references. Each of these additions is achieved while still maintaining a simplified syntax that a user can learn in a matter of minutes. It is hoped that these additions to COURSEWRITER will result in a reduction in the amount of time to program a lesson, a more thorough set of possible answer matches in a lesson, and a more convenient method of entering material into that lesson.

As stated previously, COURSEWRITER requires a single statement for each of the anticipated student responses. This limits the scope of the answers that the author may wish to take into account if he/she must program numerous responses each with explanatory text. One solution to the problem is to limit the question such that only a few responses are probable. This results in narrow questions that students will most likely answer correctly. It also requires that the lesson author work at constraining the question so that it can be programmed in a reasonable period of time. This means that if a COURSEWRITER III system is all that is available, CAI will most likely not be used extensively. This is due to the difficulty in lesson writing imposed by the construction of the language itself.

As an example, consider a CAI course on English sentence construction that asks a student to pick one word from several columns of words and form a correct sentence. The question presented might be as follows:

```
Choose one word from each column and give a correct sentence
in the form of adjective-noun-verb-adverb
   badly       house     is        a         painted
   poorly      wall      was       this      colored
   well                            that
```

For each correct answer, the author might want the message "VERY GOOD, THAT'S CORRECT" typed to the student. Additionally, the author might want to send a message telling the student when the past or present tense of a verb was used in the answer. There are 72 possible correct answers of the form adjective-noun-verb-adverb.

Programming such a question in some of the other available author languages is not very difficult. Figure V.1 shows how this question would look programmed in the GIZMO programming language. However, this same example in COURSEWRITER III becomes devastatingly time consuming. All 72 correct answers and the 72 incorrect answers must be enumerated including the text that should be typed to the student in each case. As a result, the 19 line program on GIZMO becomes a 365 line program on the COURSEWRITER III system. Figure V.2 shows a partial listing of the program as it would appear if programmed in COURSEWRITER. It should be noted here that using some of the features of the system (macro definitions and calls, buffers), this listing could be somewhat shorter. These were left out in order to fully demonstrate the repetiviteness of the predicted answers and replies. Obviously, a lesson author attempting to teach sentence structure by asking several questions of the example form cannot use COURSEWRITER unless he/she wishes to dedicate a career to entering a program. One can also see that not only would entering all the information be time consuming, but it would be extremely boring too.

There are a variety of other applications in which such burdens are placed upon the lesson author, that implementation of a lesson on COURSEWRITER is simply not possible or convenient. These areas include teaching of spelling, introductory computer programming, chemistry, or any area that may have many similar predicted responses all of which the author must specify. The SLATE system was developed in an effort to overcome some of the inadequacies inherent in the language. Certainly not all the problems in the COURSEWRITER system

```
QUESTION # AND QUESTION:
1.  CHOOSE A WORD FROM EACH OF THE COLUMNS AND FORM A CORRECT SENTENCE
    IN THE FORM:  ADJECTIVE-NOUN-VERB-ADVERB.

BADLY   HOUSE     IS        A         PAINTED
POORLY  WALL      WAS       THIS      COLORED
WELL                        THAT

GIVE A RIGHT ANSWER
<ADJ><NOUN><VERBS><VERB><ADVERB><1>

GIVE A WRONG ANSWER
<NOUN><VERBS><VERB><ADVERB><ADJ><4>

<ADJ>::=A|THIS|THAT
<NOUN>::=HOUSE|WALL
<VERBS>::=<IS>|<WAS>
<VERB>::=POORLY|BADLY|WELL
<1>::=VERY GOOD, THAT'S CORRECT.
<2>::=NOTICE YOU USED THE PAST TENSE.
<3>::=YOU HAVE USED THE PRESENT TENSE.
<4>::=THAT IS A REALLY MIXED UP SENTENCE.
<IS>::=IS<3>
<WAS>::=WAS<2>
```

Figure V.1 Demonstration of GIZMO lesson for sentence structure (underlined text is supplied by computer).

```
CA A WALL IS COLORED POORLY
TY YOU HAVE USED THE PRESENT TENSE.
TY VERY GOOD, THAT'S CORRECT.
CA A WALL IS COLORED WELL
TY YOU HAVE USED THE PRESENT TENSE.
TY VERY GOOD, THAT'S CORRECT.
CA A WALL WAS PAINTED BADLY
TY NOTICE YOU USED THE PAST TENSE.
TY VERY GOOD, THAT'S CORRECT.
CA A WALL WAS PAINTED POORLY
TY NOTICE YOU USED THE PAST TENSE.
TY VERY GOOD, THAT'S CORRECT.
CA A WALL WAS PAINTED WELL
TY NOTICE YOU USED THE PAST TENSE.
TY VERY GOOD, THAT'S CORRECT.
CA A WALL WAS COLORED BADLY
TY NOTICE YOU USED THE PAST TENSE.
TY VERY GOOD, THAT'S CORRECT.
CA A WALL WAS COLORED POORLY
TY NOTICE YOU USED THE PAST TENSE.
TY VERY GOOD, THAT'S CORRECT.
CA A WALL WAS COLORED WELL
TY NOTICE YOU USED THE PAST TENSE.
TY VERY GOOD, THAT'S CORRECT.
CA THIS HOUSE IS PAINTED BADLY
TY YOU HAVE USED THE PRESENT TENSE.
TY VERY GOOD, THAT'S CORRECT.
CA THIS HOUSE IS PAINTED POORLY
TY YOU HAVE USED THE PRESENT TENSE.
TY VERY GOOD, THAT'S CORRECT.
CA THIS HOUSE IS PAINTED WELL
TY YOU HAVE USED THE PRESENT TENSE.
TY VERY GOOD, THAT'S CORRECT.
CA THIS HOUSE IS COLORED BADLY
TY YOU HAVE USED THE PRESENT TENSE.
TY VERY GOOD, THAT'S CORRECT
CA THIS HOUSE IS COLORED POORLY
TY YOU HAVE USED THE PRESENT TENSE.
TY VERY GOOD, THAT'S CORRECT.
```

Figure V.2 Listing of some of the COURSEWRITER needed to program sentence structure question.

can be solved by SLATE. Resolution of system problems to accommodate display panels, heat sensitive response mechanisms, or to perform answer matching by more powerful algorithms can only be accomplished by a complete rewrite of the COURSEWRITER III system. What can be solved, though, is the enumeration of multiple anticipated answers with a minimal number of statements. The SLATE system can perform this task by using the computer's resources to construct a considerable portion of the author's program for him/her. Thus, lesson material that was formerly deemed not able to be implemented in COURSEWRITER, can now be programmed quite easily.

SLATE is a non-interactive sequential record processor that takes symbolic text of one of 15 opcodes and creates valid COURSEWRITER code from it. One character string of the source text may represent several hundred actual statements in COURSEWRITER. The operations being performed are similar to those of the GIZMO system.

Using SLATE, implementation of the question on page 52 concerning sentence structure is now reasonable. Figure V.3 shows the instructions that would be used to generate all the COURSEWRITER code of which a subset is shown in Figure V.2 The program is notably shorter than the original COURSEWRITER III program and the syntax closely parallels that of GIZMO. A complete explanation of the SLATE form is discussed in the next section.

A. Language Constructs

COURSEWRITER III is a subset of the SLATE language. That is, any valid COURSEWRITER statement is also a valid SLATE statement. However, it is the extensions to COURSEWRITER with which SLATE primarily is concerned.

```
QU CHOOSE A WORD FROM EACH OF THE COLUMNS AND FORM A CORRECT#
   SENTENCE IN THE FORM OF ADJECTIVE-NOUN-VERB-ADVERB*#
    BADLY     HOUSE     IS        A         PAINTED#
    POORLY    WALL      WAS       THIS      COLORED#
    WELL                          THAT
$1 A|THIS|THAT
$2 HOUSE|WALL
$3 IS<3>|WAS<2>
$4 (PAINT|COLOR)ED
$5 (BAD|POOR)LY|WELL
<1 TY VERY GOOD, THAT'S CORRECT.#
<2 TY NOTICE YOU USED THE PAST TENSE.#
<3 TY YOU HAVE USED THE PRESENT TENSE.#
<4 TY THAT IS A REALLY MIXED UP SENTENCE.#
CA ($1) ($2) ($3) $4 ($5)<1>#
WA ($2) ($3) $4 ($5) ($1)<4>#
END
```

Figure V.3 SLATE instructions to generate entire COURSE-WRITER code for lesson on sentence structure.

There are fifteen opcodes that will invoke the text expansion. These fifteen opcodes are CA, CB, WA, WB, AA, AB, UN, PR, CA(, CB(, WA(, WB(, AA(, AB(, PR(where the opcodes used with a parenthesis are opcodes followed by modifiers. The text of these opcodes is usually in a Backus-Naur like form in which there are two levels of delimiting. The basic symbols used are: "( )", major opcode delimeters, "< >", minor opcode delimeters, ":", the "and" symbol for minor opcodes, and "|", meaning "or" in a string. There are other symbols that will be discussed later.

As a simple example demonstrating the use of these symbols, suppose a teacher programs

```
CA ($1)<1:2>
```

This tells SLATE to locate the definitions of the designated numerical symbols and make the appropriate substitutions of the definitions into the text. Then, SLATE should begin expanding the code into COURSEWRITER statements. Suppose $1 has been defined to be the string

```
A|B
```

<1> is defined as

```
TY GOOD
```

and <2> is defined as

```
AD 1/C1
```

Upon encountering the example string, SLATE would generate

```
CA A
TY GOOD
AD 1/C1
CA B
TY GOOD
AD 1/C1
```

The effect is that if the student enters either "A" or "B", "GOOD" is typed by the system and 1 is added to counter 1.

The general form of any statement that can be processed is

$$MA\ (E_{11}|E_{12}|\cdots|E_{1j_1})(E_{21}|E_{22}|\cdots|E_{2j_2})\cdots(E_{m1}|E_{m2}|\cdots|E_{mj_n})$$
$$<MI_1:MI_2:\cdots:MI_p>$$

Where MA represents one of the 15 opcodes, E's represent expressions which may be either simple character strings or "or" expressions themselves, MI's represent minor opcodes that should be executed when the major opcode is executed. SLATE examines the input string and generates the following sequence of code:

$$MA\ E_{1i_1}E_{2i_2}E_{3i_3}\cdots E_{mi_n}$$
$$MI_1$$
$$MI_2$$
$$MI_3$$
$$\vdots$$
$$MI_p$$

for all $1 \le i_k \le j_k$. In words, this amounts to taking all permutations of the parenthesized expressions and following each by the specified minor opcodes contained in the brackets.

Text of any kind may precede and follow an "or" group (a sequence of "or" expressions) such that it is included in the expanded character string. Thus,

```
WA CAPIT(O|A)L
```

results in the generation of

```
WA CAPITOL
WA CAPITAL
```

It may also be convenient and is permissible to nest "or" expressions to any level. An example of a nested expression is

```
WA ADD (1(;|.)|2)
```

After the string has been processed, the student may enter

```
ADD 1;
ADD 1.
ADD 2
```

and COURSEWRITER will consider the response as a match of the designated wrong answer.

Any expression may be null or contain a null entry. Three examples of null expressions are

```
CA $S(|TART)

CB FA(R|)THER

WB ME(E||A)T
```

When the null expression is encountered, the empty string is concatenated to the string that has been created up to that position.

So far, minor opcodes have for the most part been left out of the discussion. Although their presence is optional, the lesson author usually wants to either congratulate the student after being correct or explain to the student why he/she is incorrect otherwise. Therefore, minor opcodes are usually used following each major opcode, and are thus included as part of the SLATE acceptable input string.

Minor opcodes are enclosed in brackets and, if more than one, separated by a ":". The placement in the string determines which of the major opcodes it will follow. Therefore, the string

```
CA (A|B)<TY GOOD>
```

generates

```
CA A
TY GOOD
CA B
TY GOOD
```

while

```
CA (A<TY GOOD>|B)
```

generates

```
CA A
TY GOOD
CA B
```

Thus, the lesson author, by strategic placement of the minor opcodes can control exactly when minor opcodes will be executed. Any minor opcode that appears outside of an "or" group will be included in every one of the generated major opcode groups. Any minor opcode that is placed within an "or" expression will only be included in the major opcode group whose text field contains the corresponding expression. It is important to note here that strict adherence to the rules for construction of well formed COURSEWRITER statements is requisite. This is because SLATE accepts bracketed strings exactly as they appear. For example, if

```
<TYGOOD>
```

appears in the SLATE input,

```
TYGOOD
```

will be sent to the output file and COURSEWRITER will flag the opcode as an error because of the missing space.

Frequently, the same minor opcodes are used repeatedly throughout a COURSEWRITER lesson. It becomes a tiresome job for the lesson

author to continually enter the same sequence of instructions (for example,

```
TY VERY GOOD
AD 1/C10
LD 0/S1
```

for every correct answer that might require the above instructions). Even when multiple answers are represented as SLATE instructions, such as

```
CA CAPIT(0|A)L<TY VERY GOOD:AD 1/C10:LD 0/S1>
```

the bracketed string is complex and requires careful typing. If the author used this sequence of minor opcodes often, he/she would have to type the string again and again.

An alternate method exists for expressing minor opcodes and text which applies to all minor opcodes. This method allows individual minor opcodes and associated text to be designated by a Predefined Minor Opcode (PMO) number n, where $0 \leq n \leq 31$. When a number is encountered in the bracketed string, it is replaced by the entire opcode and text that it represents. The definition of a PMO is accomplished using a new opcode. This opcode consists of a "<" and a number. As in every COURSEWRITER instruction, the opcode is always three characters in length and text may be continued over several statements using the continuation opcode of three blanks. If the number is two places, no space is skipped before beginning the definition. With the PMO, then, the previous opcodes

```
TY VERY GOOD
AD 1/C10
LD 0/S1
```

could be identified as three PMO's:

```
<1 TY VERY GOOD
<2 AD 1/C10
<3 LD 0/S1
```

The text of a PMO must also conform to rules of COURSEWRITER and may be only 72 characters in length. The correct answer could now be written

```
CA CAPIT(A|0)L<1:2:3>
```

which is much easier to type. Once a PMO has been defined, it retains that definition until the program terminates. PMO's may thus be used repeatedly without redefinition.

The lesson author may choose to integrate valid COURSEWRITER and PMO's in the bracketed strings. This may be performed easily by simply listing each PMO and each valid minor opcode with a colon For example,

```
CA CAPIT(O|A)L<1:LD BO/B1:2:3:BR PR>
```

would be expanded to

```
CA CAPITOL
TY VERY GOOD
LD BO/B1
AD 1/C10
LD O/S1
BR PR
 :
 :
```

It is not necessary that an "or" exist for code generation to occur. If the major opcode is one of the fifteen mentioned, then a bracketed string will tell SLATE to split the substrings and make the appropriate substitution. For example, the instruction

```
PR<LD O/CO-30;LD O/PO-31;LD O/SO-31>
```

would cause SLATE to generate:

```
PR
LD 0/C0-30
LD 0/P0-31
LD 0/S0-31
```

In the GIZMO system, predicted answers are given as a sequence of symbols that sometimes define "or" -ed expressions. This same kind of symbolic representation of strings is accomplished in SLATE by using the Predefined Major Expression (PME). The PME allows the user to define phrases, expressions, multiple lines of code, other PME's, or any combination of these with a single symbol. Whereas a PMO is represented by a number, the PME is represented by a "$" followed by a number. The construction of the PME is similar to the PMO in that the opcode is three characters in length and the text begins in the fourth position. A string designated by a "$n" is replaced immediately by its definition when encountered and can be used in the text field of any of the 15 major opcodes. It too may be continued over any number of cards using the continuation opcode and may be up to 255 characters in length. Using the PME, a predicted answer can thus be reduced to a totally symbolic form.

As another simple example, suppose the correct answer to a question is "computer", and predicted wrong answers are "computer", "computir", "computar", and "computur". Figure V.4 shows the SLATE code needed to construct an answer group that types "GOOD" to the student for a correct response and "THAT'S INCORRECT" for a wrong answer. The strings that are being represented are given after the

```
$1 COMPUT
$2 A|I|O|U
$3 R
<1 TY GOOD
<2 TY THAT'S INCORRECT
CA $1E$3<1>
WA $1($2)$3<2>
```

Intermediate Strings:

```
CA COMPUTER<TY GOOD>

WA COMPUT(A|I|O|U)R<TY THAT'S INCORRECT>
```

Figure V.4 SLATE code using Predefined Major Expressions and Predefined Minor Opcodes.

actual code. Obviously, using the PME in this case results in more work for the lesson author. However in more complex situations, the PME can save time. This is particularly true when the same expression is used many times in several text fields of major opcodes. As with the PMO,a symbol once defined retains its value throughout the program's execution unless redefined.

There are other characters that have significance in the SLATE system. These characters are used to identify hexadecimal control characters for the output medium of the COURSEWRITER system. The control characters are placed in the text field (usually the last character) and are used for any COURSEWRITER opcode where appropriate.

The "*" tells SLATE that the author wants the typewriter to skip to the next line, but not return the printer carriage to print position one on the page. The number of "*"'s used represents the number of line feeds to perform. Placement of this character is critical since when COURSEWRITER encounters a line feed while outputing text, it will skip to the next line immediately before continuing print.

The "¬" is used when the lesson author wants the carriage to back-space a character. Multiple "¬" signs will cause the same number of backspaces. This is sometimes used for underlining output but only works on a typewriter output device. If used on a CRT display screen, the backspace characters appear as strings of "*"'s on the screen.

The "#" indicates that the output device should do a carriage return before continuing. If the printer encounters the carriage return character, it skips to the next line and returns the carriage to column one on the page. Therefore, the lesson author does not have to specify both a line feed character and a carriage return character on statements that are processed by SLATE. The lesson author should recognize when carriage returns are needed and that even if the opcode is not one of the fifteen that SLATE processes, carriage returns must be specified. Also, at most one carriage return character may be specified per COURSEWRITER statement.

SLATE, in order to tell which character string should be processed in what way, uses eight characters as delimeters and three special control characters. Unfortunately, these use up most of the special characters on a typewriter keyboard. Often, a lesson writer may need one or more of these special characters as either part of the question or part of the answer. SLATE has no facility for identifying semantic usage of a character, but it does have an opcode that allows the user to redefine delimeters as necessary. The RE (redefine) opcode tells SLATE that a certain delimeter or character is to be replaced by another. Its form is

RE keyword1=x_1,keyword2=x_2,...

where the x's are the new characters. All keywords have a default value in SLATE as shown in Table IV. The keywords are a three letter abbreviation followed by an "=" sign. Only those keywords

TABLE IV

LISTING OF SLATE KEYWORDS, MEANINGS,
AND DEFAULT VALUES

Abbreviation	Meaning	Default
LMA	left major delimeter	(
RMA	right major delimeter	)
LMI	left minor delimeter	<
RMI	right minor delimeter	>
POR	processor "or" symbol	|
AND	processor "and" symbol	:
PMO	predefined minor opcode	<
PME	predefined major expression	$
LFD	line feed character	*
BSP	backspace character	¬
CRT	carriage return character	#

that are being redefined need be specified on an RE statement. Examples of the usage are given below:

RE LMA=+	set the left major delimeter to a "+"
RE CRT=-,LFD=@,RMI=.	the carriage return character is now a "-", the line feed is now a "@", and the right minor delimeter is a "."

Delimeters and control characters may be redefined at any time to fit the author's needs.

B. Using SLATE

Now that the SLATE syntax has been described, consider what is actually taking place in the sequence of code in Figure V.3 on page 57 for the question on sentence structure. The first statements represent the question that should be typed to the student. The "#"'s have been added to indicate that after each line is printed, the printer should return to column 1 and skip a line. The only processing that would occur for each statement in the question is that the control character would be translated to its corresponding hexadecimal character.

Lines six through ten define PME's that represent "or" expressions and will be used later in the correct and incorrect answers. That is, each time "$1" is used, "A|THIS|THAT" will be substituted. The statements eleven through fourteen define PMO's representing messages to be typed to the student if the student's response matches the predicted response.

The correct answer shows all the symbols to be used with the minor opcodes properly enclosed in brackets. A carriage return character is added to every generated string because the student's answer will contain one. All PME's have been properly enclosed in parentheses. It could have been that the parentheses were placed in the text of the PME, though. The symbolic representation of 72 correct answers could also have been written:

```
CA (A|THIS|THAT)(HOUSE|WALL)(IS<TY YOU HAVE USED THE PRESENT
   TENSE>|WAS<TY NOTICE YOU USED THE PAST TENSE>)(PAINT|COLOR)
   ED((POOR|BAD)LY|WELL)<TY VERY GOOD, THAT'S CORRECT.>#
```

If the author had chosen this form, statements six through fourteen could have been eliminated. However, in the wrong answer, the same strings would have had to have been rewritten to achieve the necessary answer repertoire.

This short program would result in the expansion of the correct and incorrect answers to look like the figure on page 55. It brings the question-answer sequence into the realm of implementation. Asking a lesson writer to write a sixteen statement program is not unreasonable.

The SLATE system is designed to accept any 71 (or more) character input dataset and process it sequentially while creating two output files. The first output file is a printed listing of all the data read and the action taken on each statement. The second output file is written to a permanent storage device (usually magnetic tape) that contains the processed COURSEWRITER program. This is done because COURSEWRITER has no facility for a direct insertion of the code into its own data files by a program running

external to the system. Therefore, the insertion process is performed by the AUTO INSERT function within the COURSEWRITER system. This function takes the program that is on tape and forms chained variable length records that in turn are stored on disk. The process must be run when COURSEWRITER is not operational and depending on the length of the program, takes anywhere from a few seconds to several minutes of CPU time. After the AUTO INSERT has been run, the COURSEWRITER program is available for student use.

To use SLATE, a lesson author writes the program as if he/she were actually working at a terminal connected to COURSEWRITER. The program is usually punched on 80 column cards with the opcode in card columns 1 through 3 and the text, if there is any, in columns 4 through 71. AUTO INSERT normally requires a "SIGN ON" card but this may be deleted since one is automatically generated by SLATE. The first card required of the author is the course name. This starts in column 1 and contains no other characters except those in the name. The second card required is the "INSERT AFTER" statement that tells the system where to place the statements that follow. This also starts in column 1 and is of the form:

INSERT AFTER label-problem-sequence-number

with no control characters added.

After these two cards, the author places the COURSEWRITER program which will be processed by SLATE creating the new COURSEWRITER dataset.

As each record is read, SLATE checks to see if the opcode is one of fifteen that will allow the text to be expanded, a definition of a symbol, or a redefinition of a delimeter. If none of these conditions are met, then the statement is assumed to be valid COURSEWRITER code and is passed to both output files with two additions. The first addition is an end-of-block character (EOB). All COURSEWRITER statements must have this hexadecimal character as the last character in the string. The user is not responsible for having this character in the string as it is automatically added. The EOB tells COURSEWRITER that everything that precedes it should be processed as part of the instruction. It is for this reason that only 71 characters are read from a card. The EOB makes 72 which AUTO INSERT processes. The EOB is not printed on the output since it is a non-printable character.

The second addition to the output string is a sequence number that begins at 1 and may reach 32767. This sequence number is also written to the output tape as a card image in columns 73 through 80 and is used for identification purposes by AUTO INSERT. If AUTO INSERT finds an invalid statement, it writes a message to the command console for the system specifying the type of error and the card number where the error occurred. The lesson

author can, then, look at the listing, find the statement with that sequence number, and re-insert the correct statement by signing on a terminal later. If AUTO INSERT does find an error, the erroneous statement is skipped.

An opcode that results in text expansion causes SLATE to generate the appropriate code enclosed in a box on the printed output. That statement contains no sequence number, but the generated statements do. Each generated statement also contains the EOB character. Thus, if a programmer writes

```
PR(R)<AD 1/C1:LD 0/P0>
```

the program listing would look like

```
PR(R)<AD 1/C1:LD 0/P0>
*************************************************
|   *PR(R)                              00000001  |
|   *AD 1/C1                            00000002  |
|   *LD 0/P0                            00000003  |
*-----------------------------------------------*
|    3 STATEMENTS GENERATED                       |
*************************************************
```

This shows the author exactly what was generated from the input string. The "*" that precedes each statement signifies that the instruction was generated. The tape output file contains 80 character records which contain the boxed statements.

If the opcode denotes a predefinition of a symbol, then that statement does not receive a sequence number or an EOB. The statement is printed but is not sent to the tape file since that opcode is not a valid COURSEWRITER instruction. The text is stored in the symbol table exactly as it appears and SLATE reads the next card to be processed.

If an RE opcode is found, then SLATE redefines the delimeter as specified and outputs to the print file the status of every keyword. The opcode has no effect on the tape output and therefore receives no sequence number.

The last card of the program is the "END" card. The word "END" is placed in the opcode field and signifies the end of the program. When encountered, SLATE sends it to both output files since it is required by AUTO INSERT.

Only one "END" statement may appear in any SLATE program. This is because AUTO INSERT will only accept one group of statements for each "INSERT AFTER" operation. If more than one group of statements is to be processed, SLATE should be run for each group. Appendix C shows all of the necessary job control language necessary to accomplish single or multiple runs within one job stream for processing on the IBM 360/65 at the University of Tennessee.

When all cards have been read, summary statistics are printed stating the total number of cards read, the total number of statements generated, the total number of COURSEWRITER statements in the program, and the total number of errors encountered while processing. In addition, since the printed listing is sometimes difficult to read because of all the information printed by SLATE, a second listing is produced that shows the entire processed program as it will be given to COURSEWRITER through AUTO INSERT. This output includes the sequence numbers for each statement. At this point processing terminates.

C. Physical Requirements of System

SLATE is written in PL/I and requires a minimum of 88K of processor storage. This amount will increase somewhat depending on the quantity of recursions done in the system. For very complex statement constructions, the system usually needs approximately 100K to perform the text expansion. More core may be required if there are several nestings of "or" expressions but rarely should requirements reach 128K.

SLATE's processing speed is adequate. Empirically derived results show that the average run time exceeds 3200 characters per second. This means that a 20 to 30 line SLATE program could result in a 1000 statement COURSEWRITER program with approximately 10 seconds of processing time. Therefore, the costs in computer time certainly offset the costs of programmer time.

One of the major problems that SLATE has is related to the COURSEWRITER system itself. All the enumerated answers consume excessive direct access storage space. Currently, the COURSEWRITER III system resides on an IBM 2314 disk storage device. The system also places all of its datasets (the CAI courses) on 2314's in blocks of 360. Though a user may have several hundred blocks allocated to his/her course, it would probably not be long before several questions in SLATE format could result in course allocation being consumed with predicted answers. Larger disk packs would be necessitated in order to use SLATE adequately. For example, the IBM 3330 disk drive could be used which allows storage of up to 100

million characters compared to 29 million on the 2314. If a true CAI laboratory using COURSEWRITER existed, it is likely that entire disk packs would be allocated to individual courses, thus, overcoming the storage problem. Also, when a COURSEWRITER program is finally completed, it can be optimized by the COURSEWRITER system such that storage space can be reduced by as much as 30 percent.

CHAPTER VI

INTERNAL STRUCTURE AND OPERATION

The SLATE system is comprised of a main procedure and twelve subroutines that perform the string processing. The basic flow diagram of the system is given Figure VI.1. As shown, SLATE reads an opcode and text and determines in what class the opcode belongs. This section will describe which tasks the main procedure and twelve subroutines are assigned and which routines are executed depending on the opcode classification.

After an opcode and text are read, the trailing blanks are trimmed from the record. If the entire text is blank, it is considered null. Following this action, SLATE begins testing the opcode. Since there are fifteen possible opcodes that can invoke text expansion, the main procedure performs a loop that attempts to match the opcode with one of the fifteen. If a match is found, SLATE calls a routine (called READNEXT in the actual code) to read the next card, determine if that card is a continuation, and concatenate the contents of the card if its opcode is three blanks. Continuations may be over any number of cards in any manor because READNEXT halts only when a nonblank opcode is encountered. The maximum string length is 255 characters.

When the entire string has been read, SLATE checks for any symbols that might represent PME's ($number). If one is found then a subprogram (called REPLAC in the actual code) makes the necessary

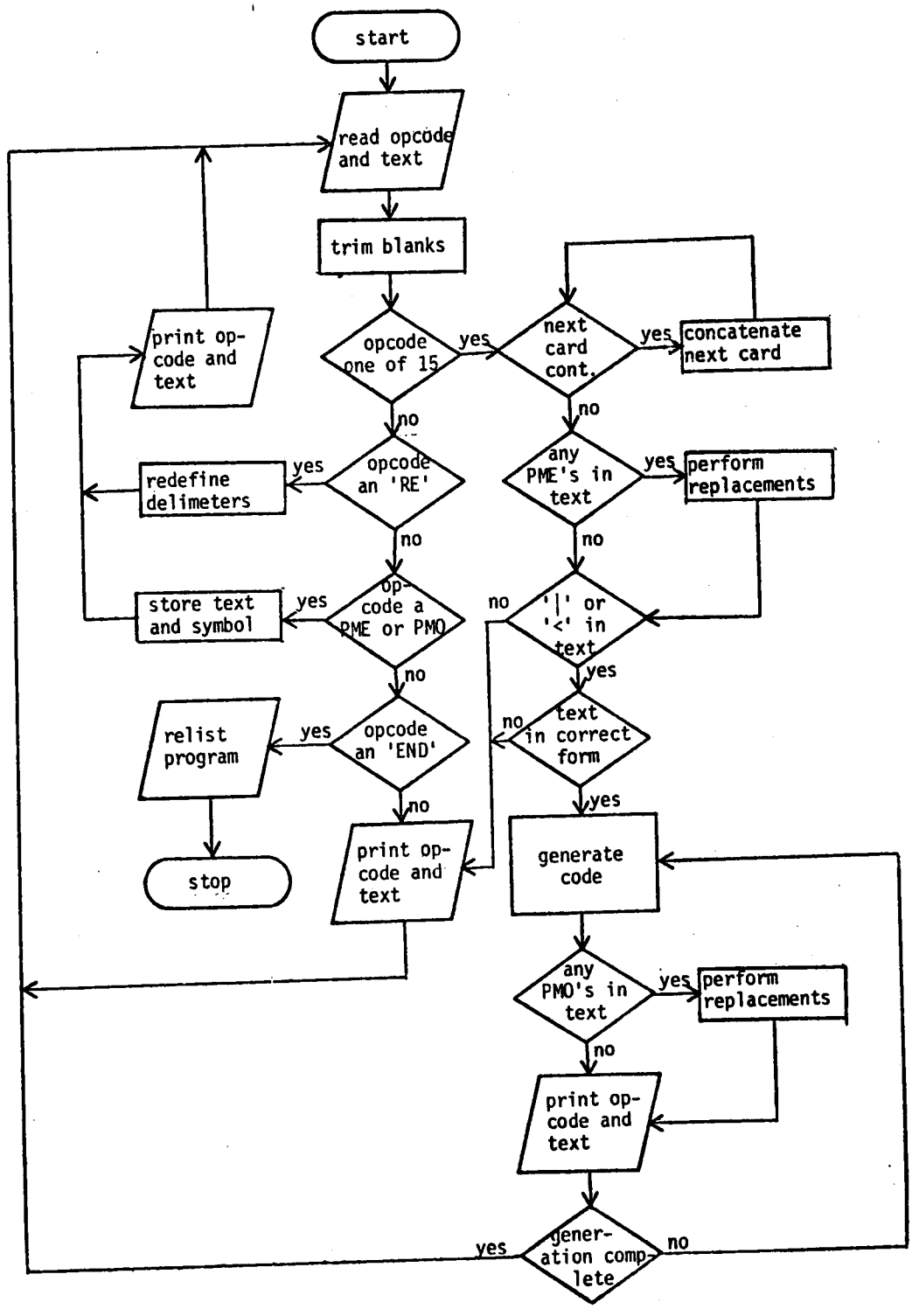

Figure VI.1 Flow diagram of the SLATE processing system.

replacements. It performs this task by finding the appropriate symbol in the text string, comparing each symbol in the table to the designated symbol, and, upon a match, inserting the text from the corresponding entry from the text table into the string. After the replacement, the string is re-examined and the process repeated if necessary. See Figure VI.2. If more than 255 replacements are attempted, an error is printed because this occurrence probably indicates that the user has recursively defined a symbol.

Upon completion of the replacement, SLATE proceeds to check the text field for the "or" symbol or brackets if there is no "or" (performed by routine ORCHK). If there is one present, a syntax check is made on the text. The routine VALCHK examines each character of the string checking for such errors as unbalanced parentheses and brackets. One of three condition codes can be returned by VALCHK.

1. The string has unbalanced parentheses (major delimeters).

2. The string has balanced parentheses but unbalanced brackets (minor delimeters).

3. The string has no errors.

If condition one exists, then the appropriate error message is printed and the statement is passed to both output files. Condition two results in text expansion but only to the major level. In other words, text enclosed in brackets is left unexpanded (It is possible the lesson author may want this to happen for the instance when the "<" should be included as part of the anticipated answer.). An

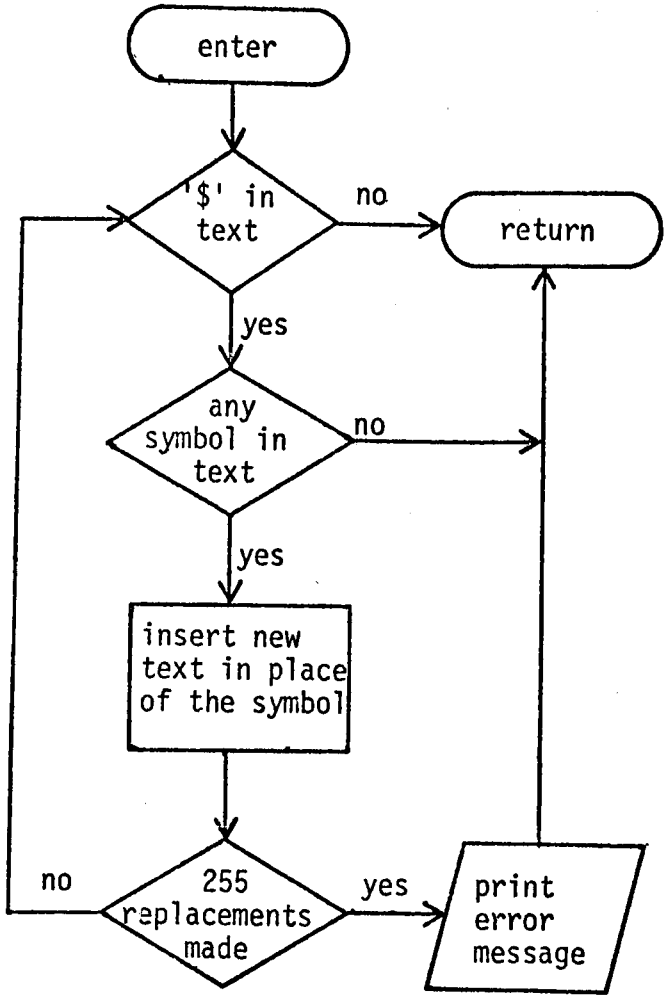

Figure VI.2 Procedure for replacement of PME's in text of SLATE instructions.

error message will be printed should this occur. Condition three results in normal processing and the code will be generated as usual. See Figure VI.3.

The routine GENCAI is where most of the code expansion takes place. In conjunction with three other subroutines (LOCATE, SEPRAT, and PRINT), the recursive routine performs the generation as diagrammed in Figure VI.4. The algorithm used is as follows:

ALGORITHM GENCAI: This is a scanning technique that generates COURSEWRITER strings from a given input string with "or" expressions. In this algorithm, SNTNCE is the completely generated string, BEGIN is all characters up to the first "or" group, PART is the "or" group itself, WORD is the individual string from the "or" group, REST is the string of characters that follow the "or" group (may contain "or" groups also), LEN1 is the length of WORD+2 (to get position of the next word in the "or" group), and LEN2 is the character length of the "or" group excluding parentheses.

STEP G1: [Initialization] PART ← position of first character of a string within the "or" group; BEGIN ← string up to an "or" group; REST ← everything following the "or" group.

STEP G2: [Get individual string from "or" group] WORD ← a string from the "or" group.

STEP G3: [Form sentence] SNTNCE ← BEGIN||WORD||REST

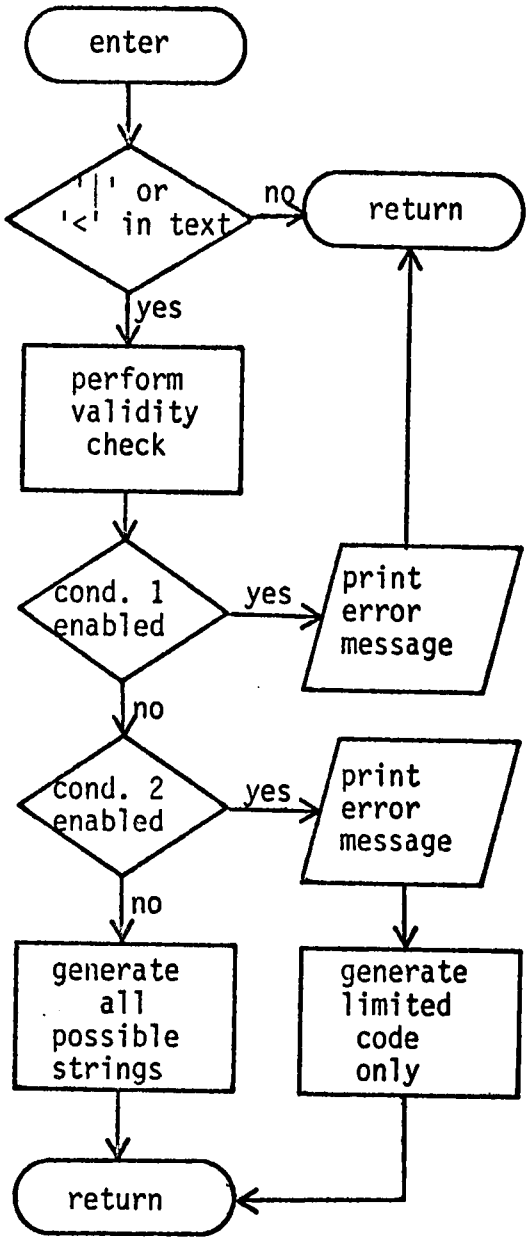

Figure VI.3 Routine ORCHK flow diagram.

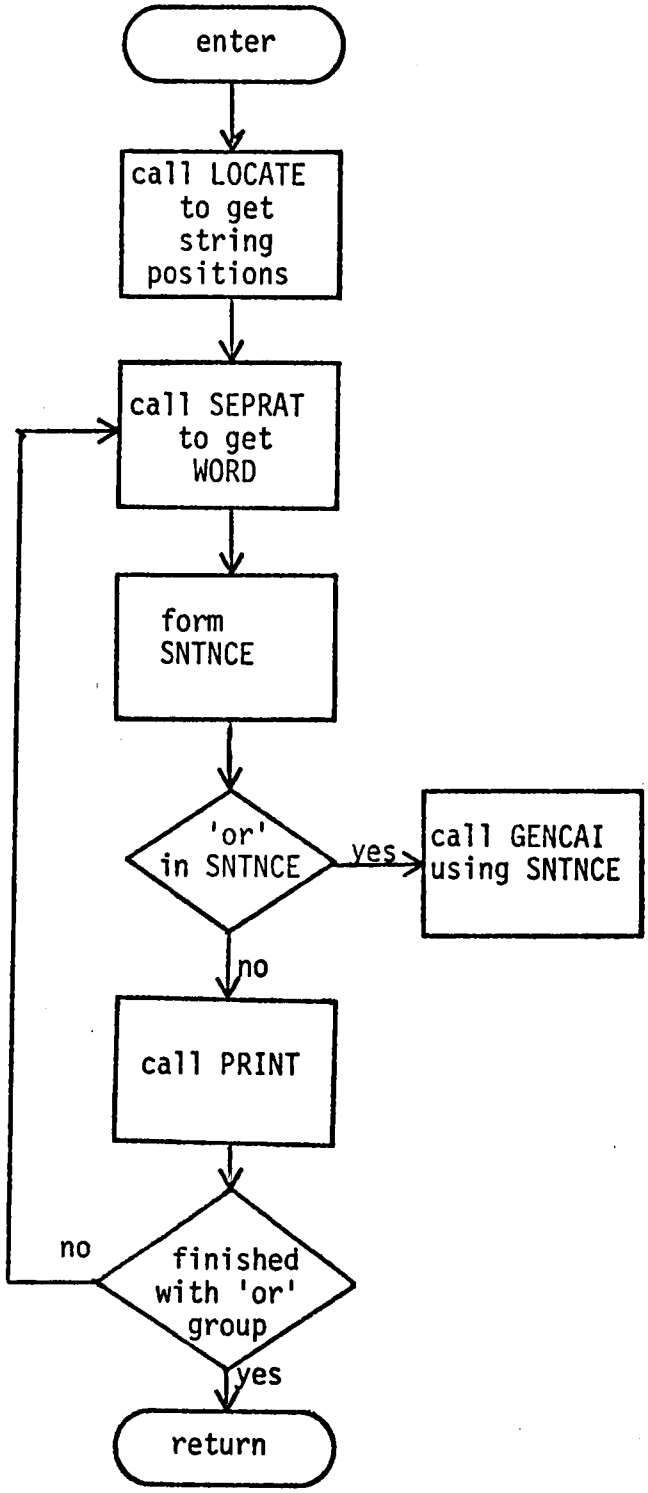

Figure VI.4 Recursive algorithm for COURSEWRITER code generation.

STEP G4: [Check for "or" sign in new sentence] If SNTNCE contains an "or" sign then recurse using SNTNCE as the input string; else call PRINT to print sentence.

STEP G5: [Reset lengths of substrings] LEN1 ← length(WORD)+2; LEN2 ← length(PART)

STEP G6: [See if null string is part of "or" group] If LEN1=LEN2, then go to step G2; else go to step G7.

STEP G7: [See if completed separating all the strings from the "or" group] If LEN1<LEN2 then PART ← next string from "or" group; else return.

Routine LOCATE returns the actual positions in the string of the "or" group and the character strings stored in BEGIN and REST to GENCAI. Routine SEPRAT recursively separates the individual strings from the "or" expression.

When the string is ready to be printed, it may possibly contain bracketed minor opcodes which must still be expanded. The routine PRINT must determine if there are minor opcodes expressed and separate them from the string. PRINT performs the separation by finding the bracketed strings and creating a new string that contains each minor opcode delimited by a ":" and deleting each minor opcode from the input string. The original opcode, text, and appended EOB can now be printed in both files. The statement is printed with a preceding "*" showing that it was generated.

Following this action, PRINT concerns itself with the string of minor opcodes. It is possible that both PMO's and valid COURSEWRITER

appear in this string. If PRINT locates a valid COURSEWRITER minor opcode, it is printed as is. If PRINT locates a PMO occurrence, then the PMO number is used as an index to the PMO table for the text that should be printed. If the PMO is undefined, blanks are substituted as the opcode since no valid COURSEWRITER opcode contains a number. When the end of the minor opcode string is reached, the subprogram returns to GENCAI to get the next statement. See Figure VI.5.

When GENCAI has created its last statement and PRINT has sent it to both output files, the routine returns to ORCHK. Here a final summary of the total number of statements generated is printed and the program returns to the main procedure to get the next opcode and text.

In terms of formal language theory, each of the parenthesized and bracketed symbols is a non-terminal that derives, in a finite number of steps, a sequence of characters inserted at the generation phase of execution. These characters represent terminal strings. The language generated by the system can be defined by the procedure derived as follows:

STEP 1: For each of the specified correct or incorrect answers generate the following productions:

$$S \rightarrow A_i$$

$$A_i \rightarrow \psi$$

where i is the number of the answer and ψ is the string of terminals and/or non-terminals.

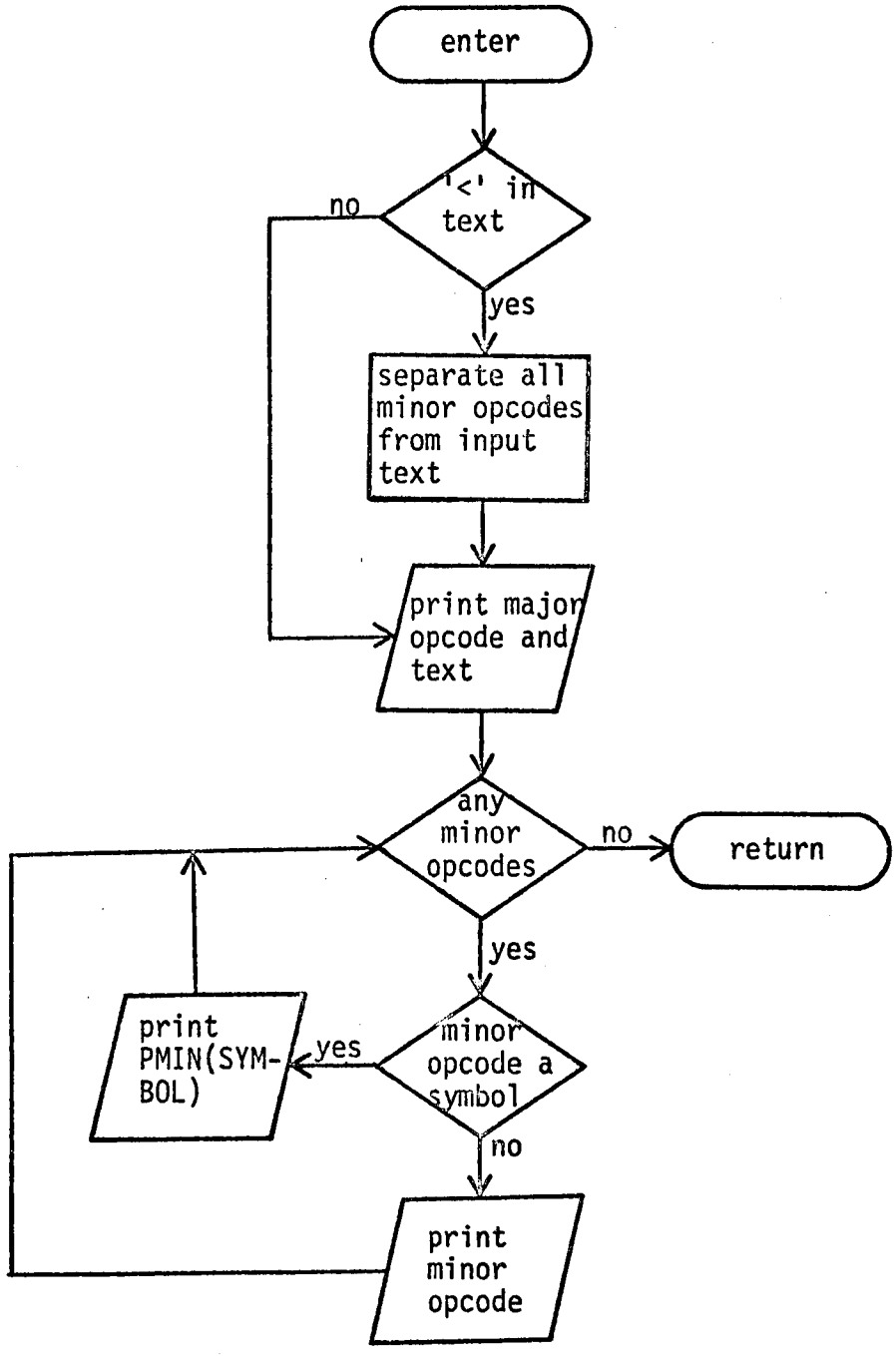

Figure VI.5 Print procedure for major and minor opcodes called by the generation routine.

STEP 2: For each definition that does not contain an "or" sign generate the production

$$A \rightarrow \psi$$

where ψ is a string of terminals and/or non-terminals.

STEP 3: For each definition of a non-terminal that contains an "or" sign generate

$$A \rightarrow \alpha_i$$

Here each α_i is a string separated by "or" signs.

This set of productions can be shown to accept regular languages. First keep all productions of the form $A \rightarrow \alpha$ since α is a string of terminals only. For each production of the form

$$A \rightarrow \psi$$

where ψ is a string of terminals and non-terminals, each non-terminal must derive a string of terminals in a finite number of steps since recursive definition is not allowed. Thus, for each non-terminal B, generate a new set of productions

$$B \rightarrow \alpha C$$
$$C \rightarrow \alpha$$

where α is a string of terminals. Now all the productions are either of the form

$$A \rightarrow \alpha$$

or

$$A \rightarrow \alpha B$$

which is indeed a regular grammar, and, thus, accepts regular languages.

Conversely, the language generated by this system is also regular. For each production of the form

$$A \rightarrow \alpha_i$$

replace α by the definition of the non-terminal A. For the productions of the form

$$A \rightarrow \alpha B$$

supply α $number ($number is the non-terminal symbol) for the definition of A. If multiple definitions result, they may all be included in the definition separated by "or" signs.

The discussion to this point has been a description and conceptualization of what SLATE does if it is a match of one of the fifteen major opcodes. If no match occurs, however, more tests are made on the opcode. If the opcode is an RE, then the lesson author wants to define the delimeters or control characters. Another subroutine (called REDEF in the actual code) makes the new assignments. In this routine a simple check is made for the occurrence of one of the eleven keywords. Although, no commas are required between keywords, they are recommended to lessen the likelihood of error. Two important error checks are made in this routine. The first is the case that the user may have defined the major delimeters to be the same symbol. The second is the case that the minor delimeters may have been defined to be the same character. Either case will result in an unrecoverable error when GENCAI accesses the string later. Therefore, should either situation arise, execution terminates.

If the opcode is not a redefinition, then a test is made to see if it is a symbol definition. For a PME, SLATE maintains two tables. The first table is a symbol table that consists of the $number stored in an array (called PMAE) location whose subscript is the number itself. Before storing this symbol, though, a routine (called CONVRT) attempts to convert this symbol to a valid decimal entry. If the symbol is not a number (excluding the "$"), then that opcode is in error, an error message is printed, and the entire statement is deleted. However, upon a successful conversion, the symbol is stored and the text that the symbol represents is stored in another array (called EXPAND) indexed by the decimal value of the symbol. See Figure VI.6. REPLAC can later make the replacement in a string by substituting EXPAND(J) where J is the symbol value.

If the opcode denotes a PMO, then the same conversion check is made upon the number as with the PME. If CONVRT fails then an error message is printed and the statement deleted. However, SLATE does not store the PMO exactly the same way as with the PME. Instead of creating another symbol table, SLATE creates a text table (PMIN) and stores the text of the PMO in PMIN(J) where J is the PMO decimal value. See Figure VI.7. Then, when PRINT encounters a symbol such as <1> in the text, it uses the contents of PMIN(1).

Upon no match with any of the above, a last check is made to see if the opcode is an "END" which will transfer the program to the relisting section. This simply rewinds the output file, CAITAPE, and performs a sequential print of each record.

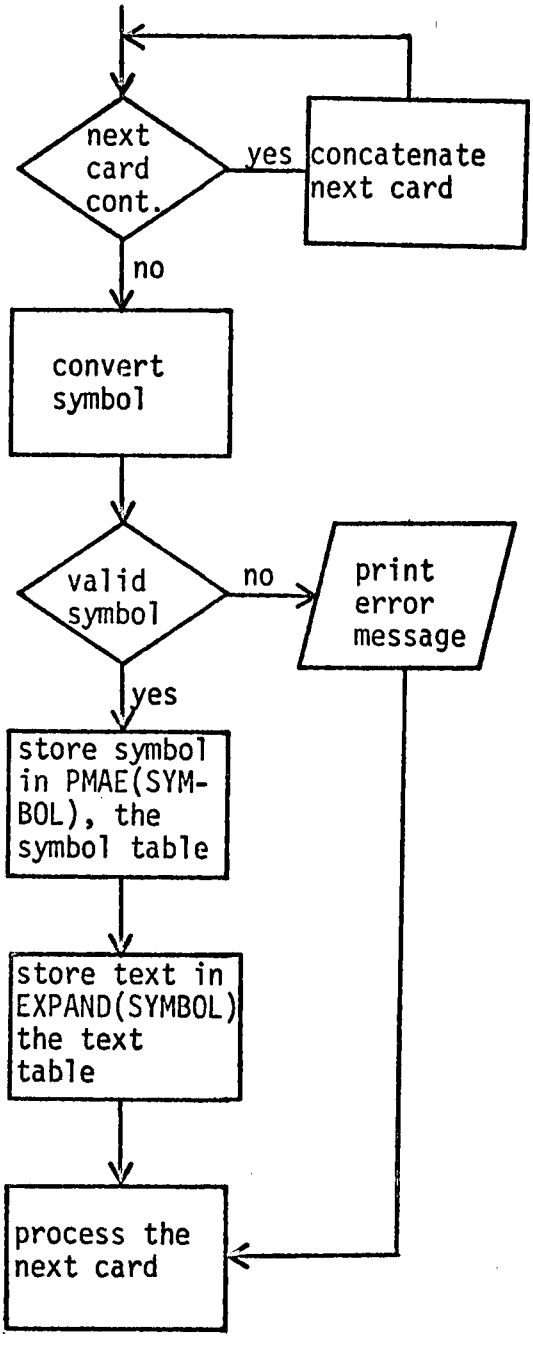

Figure VI.6 Storage technique of Predefined Major Expressions and associated text.

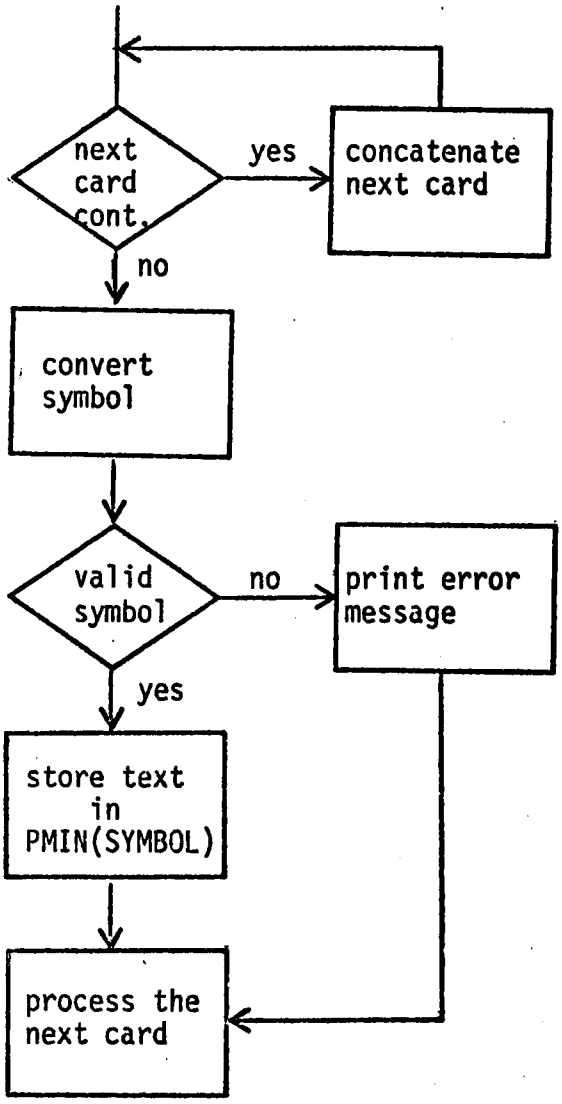

Figure VI.7 Method used for storing text of a Predefined Minor Opcode.

If the opcode is none of the significant opcodes as far as SLATE is concerned, the statement is printed and SLATE proceeds to the next record in the input stream.

This chapter has been devoted primarily to an explanation of the actual workings of the programmed system. Example usage of this system is too lengthy to present here, but two examples showing SLATE code and the actual code generated are supplied in Appendix B.

CHAPTER VII

CONCLUSION

Though SLATE achieves its basic goals, there are other problems which possibly may be solved with some extensions to SLATE. The first and most obviously needed modification to the current version is to make the system somewhat interactive. Though the problems stated in Chapter I seem insurmountable, it is likely that through compromises and concessions the necessary alterations in the operating system, COURSEWRITER, and SLATE can be made.

Online operation would represent many advantages to a lesson author. First and foremost, the lesson writer could enter an opcode and text and immediately have all the valid COURSEWRITER III code expanded and inserted into the system datasets. This would allow real time syntax analysis of the SLATE statement and the subsequent generated statements. Also, the author could switch from author mode to student mode and begin testing the course segment immediately. Less time would be lost waiting for COURSEWRITER to be shutdown and AUTO INSERT run for each course or course segment.

There are two problems that can arise with the online version, though. If the text of a SLATE instruction is syntactically correct, but not correct in terms of what the author needs, several hundred statements could be generated that are totally useless. Deletion of these statements would be ineffective because COURSEWRITER does no free disk space collection to retreive space previously used by the course. The space collection must be performed by the system supervisor

using the reassembly function in COURSEWRITER. This, like AUTO INSERT, must be performed when COURSEWRITER is not operational. Therefore, one typographical error could prove temporarily disastrous for the course author. A verification check should be included to help alleviate this problem.

The second problem evident is the problem of free space in the course disk allocation. It is not uncommon for a SLATE statement to generate many statements to be added to the course. Though not a readily obvious problem, SLATE could be generating these many statements rapidly consuming the allocated blocks of storage. The real problem arises when SLATE runs out of space and still has statements to generate. Some kind of protection feature would have to be implemented to inform the lesson author when space is running short and where SLATE quit expanding the text. SLATE would restart when more space is later allocated. It would be beneficial if the altered version could store its termination status and continue without author intervention.

Another useful addition to SLATE, which could be implemented in the offline version, is an expander for algebraic expressions of counters. Suppose the author needs a sequence of operations performed on counter values before any answer comparisons are made. In COURSEWRITER, this requires that the author break the sequence of operations down into single opcodes using those described in Chapter IV. A more natural way to express a sequence, similar to that using TUTOR on the PLATO system, is to specify the counters with arithmetic operators separating them. For example, suppose an author has

written an arithmetic drill that uses five random numbers as values for the student to add, subtract, multiply, and divide. Each of these random numbers are stored in counters (assume one through five). If the author needs to evaluate an expression such as

```
C6=C1+C2/(C3*C4)-C5+6
```

without destroying any counter values and using fixed point arithmetic, he/she must program the following:

```
LD C3/C6
MP C4/C6
DV C2/C6
AD C1/C6
SB C5/C6
AD  6/C6
```

This is a useless burden placed upon the author. A SLATE opcode, such as EV (evaluate) could be created and the statement

```
EV C6=C1+C2/(C3*C4)-C5+6
```

could be used to generate the above COURSEWRITER code. In actual implementation care would have to be taken to use only unallocated counters for additional temporary storage locations for some arithmetic operators.

COURSEWRITER has many problems (limited graphics, slow input/output, poor editing features) which lead one to consider replacing COURSEWRITER with a "better" CAI system. Unfortunately, "better" usually implies more expensive. While it is true that either PLATO or TACCIT could be a more versatile CAI system, each requires its own unique hardware and total dedication of the computer resources.

It is safe to say that most small or even large academic institutions cannot afford large capital outlays for a computer system with one function. COURSEWRITER is economical which justifies the efforts to improve it. There are 27 models of IBM computers that will support the COURSEWRITER system (Namkung, 1975). Most of these are models of the 360 and 370 which are very common computing systems. Thus, COURSEWRITER can reside on a machine that is probably already being used for administrative or educational purposes without severely decreasing the computer resources or increasing total computer costs.

If the trend toward more CAI use continued more demands are going to be placed upon the COURSEWRITER system. These demands must be met if COURSEWRITER is going to be a viable tool in course presentation. SLATE is a positive step toward satisfaction of those needs and demands.

LIST OF REFERENCES

LIST OF REFERENCES

Adams, E. N., "Issues Considered in the International School 1972," Computers in the Instructional Process, edited by Karl L. Zinn, Mario Refice, and Aldo Romano, EXTEND Publications, Ann Arbor Michigan, pp. 23-27, July, 1972.

Albert, D. and Bitzer, D. L., "Advances in Computer-Based Education," Science, vol. 167, pp. 1582-1590, March 20, 1970.

Berger, H. S., "The Pennsylvania State University Computer Assisted Instruction," Inquiry, vol. 2, pp. 2-4, June, 1967.

Bitzer, Donald L., Sherwood, Bruce Arne, and Tenczar, Paul, Computer-Based Science Education, Urbana, Illinois, Computer-Based Education Research Laboratory, May, 1973.

Butman, Robert C., "CAI--There Is a Way to Make It Pay," Educational Technology, vol. 7, pp. 5-9, December, 1973.

Crowder, Norman A., "Automatic Teaching by Intrinsic Programming," Teaching Machines and Programmed Learning, edited by A. A. Lumsdaine and Robert Glaser, National Education Association for the United States, Washington, pp. 286-298, 1960.

Davis, Alan M., and others, "Instructional Software for an Assembly Language Course Taught on a Mini-computer," Proceedings of the ACM SIGPLAN Symposium on Pedagogic Languages with Small Computers, edited by William G. Bulgren and Earl J. Schweppe, Lawrence, Kansas, pp. 236-248, 1972.

Davis, Alan M., "The Design, Implementation, and Analysis of a Computer-Assisted Instruction System on a Mini-computer," Thesis, University of Illinois, 1973.

Devos, Marina, Kessis, Jean-Jacques, and Rose, Robyn, "Developments in Computer Assisted Instruction in France," Educational Technology, vol. 5, pp. 47-49, August, 1971.

Frye, Charles H., "CAI Languages, Capabilities, and Applications," Datamation, vol. 14, pp. 34-37, September, 1968.

IBM Corporation, COURSEWRITER III Version 3 Student Text, White Plains, New York, 1971.

IBM Corporation, COURSEWRITER III Version 3 Applications Description, White Plains, New York, 1973.

IBM Corporation, COURSEWRITER III Version 3 Author's Guide, White Plains, New York, 1973.

Johnson, Robert J., "Computers in Education, An IBM Viewpoint," Educational Technology, vol. 4, pp. 16-18, December, 1971.

Koerner, James, "Educational Technology," Saturday Review of Education, vol. 1, pp. 43-45, May, 1973.

Lyman, Elizabeth R., PLATO Highlights, Urbana-Champaign, Illinois, Computer-Based Education Research Laboratory, December, 1975.

Namkung, Inja, "Computer Assisted Instruction in Higher Education," Dissertation, University of Tennessee, 1975.

Oxley, Donald Wayne, The Design, Implementation of an Educational Timesharing System for the PDP-11, University of Illinois, Urbana, Illinois, June, 1972.

Pentages, Angeline, "Control Data's Education Offering: 'Plato Would Have Enjoyed PLATO'," Datamation, vol. 22, pp. 183-187, May, 1976.

PLATO, University of Illinois, Urbana-Champaign, Illinois, Computer-Based Education Research Laboratory, 1974.

Pressey, Sidney, L., "A third and Fourth Contribution Toward the Coming 'Industrial Revolution' in Education," School and Society, vol. 36, pp. 668-672, November 19, 1932.

Sherwood, Bruce Arne, The TUTOR Language, Computer-Based Education Research Laboratory and Department of Physics, University of Illinois, Urbana, Illinois, 1972.

Skinner, B. F., "The Science of Learning and the Art of Teaching," The Harvard Educational Review, vol. 24, pp. 86-97, January-December, 1954.

Stifle, Jack, The PLATO Student Terminal, Urbana, Illinois, Computer-Based Education Research Laboratory, November, 1974.

Suppes, Patrick, "The Uses of Computers in Education," Scientific American, vol. 215, pp. 207-220, September, 1966.

The Minneapolis School Districts Data Processing Joint Board Presents TIES, Minneapolis, Minnesota, 1973.

The PLATO System, University of Illinois, Urbana-Champaign, Illinois, Computer-Based Education Research Laboratory, 1974.

Visonhaler, John F. and Bass, Ronald K., "A Summary of Ten Major Studies on CAI Drill and Practice," Educational Technology, vol. 6, pp. 29-32, July, 1972.

Volk, John L., Mason, William F., and Zranket, Charles A., An Overview of the TICCIT System, MacLean, Virginia, January, 1974.

Zinn, Karl L., "Instructional Uses of Interactive Computing Systems," Datamation, vol. 14, pp. 22-27, September, 1968.

Zinn, Karl L., "Programming Conversational Use of Computers in Instruction," Proceedings of the 23rd National Conference of the ACM, Washington, pp. 85-88, 1968.

APPENDIXES

APPENDIX A

PL/I PROGRAM LISTING

```
SLATE:  PROC OPTIONS(MAIN);


/*************************************************************************/
/*                                                                       */
/*  THIS IS A PL/I PROGRAM DESIGNED TO ASSIST A COURSEWRITER III         */
/*  LESSON WRITER IN THE ENUMERATION OF ANTICIPATED ANSWERS.  THE        */
/*  GOAL OF THE PROGRAM IS TO RELIEVE THE AUTHOR OF MANY OF THE          */
/*  TIME CONSUMING TASKS INVOLVED WHEN PROGRAMMING A COURESWRITER        */
/*  LESSON SUCH THAT THE AUTHOR MAY CONCENTRATE ON LESSON DEVELOP-       */
/*  MENT RATHER THAN ACTUAL PROGRAMMING.                                 */
/*                                                                       */
/*  THIS PROGRAM, WHICH IS A PREPROCESSOR, RUNS IN CONJUNCTION WITH      */
/*  THE AUTO INSERT FUNCTION OF THE COUSEWRITER SYSTEM.  A DATASET       */
/*  IS PRODUCED BY THE PROGRAM WHICH IS INSERTED INTO THE COURSE-        */
/*  WIRTER SYSTEM AND THEN IS AVAILABLE FOR STUDENT USE.  THE INPUT      */
/*  DATA IS IN THE FORM OF 80 BYTE RECORDS, 72 OF WHICH MAY CONT-        */
/*  AIN COURSEWRITER CODE.  THE REMAINING BYTES ARE IGNORED.  THE        */
/*  OUTPUT IS ALSO 80 BYTES, 72 OF COURSEWRITER, 8 ARE SEQUENCE NUM      */
/*  BERS.  EACH RECORD PRODUCED CONTAINS ONE CHARACTER WHICH IS AN       */
/*  END-OF-BLOCK (EOB) CHARACTER THAT IS AUTOMATICALLY ADDED.  THE       */
/*  LESSON AUTHOR IS RESPONSIBLE FOR ADDING CARRIAGE RETURNS AND         */
/*  LINE FEEDS AS NECESSARY.                                             */
/*                                                                       */
/*  ALL DOCUMENTED COURSEWRITER TEXT IS ACCEPTED BY THIS PRE-            */
/*  PROCESSOR.  THE ADDITIONS ARE THE ABILITY TO DEFINE SEVERAL          */
/*  CORRECT, WRONG, OR ANTICIPATED ANSWERS ALONG WITH ANY MINOR          */
/*  OPCODES IN ONE STATEMENT.  THIS STATEMENT MAY BE UP TO 255 CHAR-     */
/*  ACTERS LONG AND MAY BE CONTINUED OVER ANY NUMBER OF CARDS USING      */
/*  THE STANDARD CONTINUATION OPCODE OF 3 BLANKS ON THE CONTINUA-        */
/*  TION CARD.  THE SYNTAX OF A STATEMENT TO BE PREPROCESSED IS A        */
/*  MAJOR OPCODE FOLLOWED BY A BACKUS-NAUR LIKE FORM OF THE DIFFER-      */
/*  ENT POSSIBLE COMBINATIONS OPTIONALLY FOLLOWED BY ANY MINOR OP-       */
/*  CODES THAT SHOULD BE EXECUTED WHEN A MAJOR OPCODE IS EXECUTED.       */
/*  THE GENERAL FORM IS AS FOLLOWS (USING DEFAULT DELIMETERS):           */
/*                                                                       */
/*  MA TEXT (E11|E12|...|E1N) (E21|E22|...|E2N) ... (EM1|EM2|...        */
/*          |EMN) <MI1:MI2:...:MIN> TEXT                                */
/*                                                                       */
/*  WHERE MA IS ONE OF THE MAJOR OPCODES                                 */
/*        TEXT IS COURSEWRITER TEXT TO BE INCLUDED                       */
/*        E IS THE EXPRESSION THAT IS PART OF THE NEW TEXT               */
/*        (IT MAY ALSO CONTAIN 'OR-ED' EXPRESSIONS)                      */
/*        MI IS THE MINOR OPCODE THAT SHOULD BE EXECUTED ALSO            */
/*                                                                       */
/*  MINOR OPCODES MAY BE PLACED ANYWHERE WITHIN THE PARENTHESES AS       */
/*  LONG AS THEY ARE DELIMITED AS SUCH.                                  */
/*                                                                       */
/*************************************************************************/
```

```
/*********************************************************************/
/*                                                                   */
/*  AN EXAMPLE OF THE ABOVE.  IF ANY OF THE SENTENCES:               */
/*                                                                   */
/*       THE BOY IS HERE                                             */
/*       THE BOY WAS HERE                                            */
/*       THE BOY IS THERE                                            */
/*       THE BOY WAS THERE                                           */
/*                                                                   */
/*  WERE TO BE ACCEPTED AS A CORRECT ANSWER AND 'GOOD' WERE TO BE    */
/*  TYPED BACK TO THE STUDENT, THE STATEMENT TO PRODUCE ALL OF THE   */
/*  ABOVE WOULD LOOK LIKE THIS:                                      */
/*                                                                   */
/*  CA THE BOY (IS|WAS) (HERE|THERE)<TY GOOD>                        */
/*                                                                   */
/*  THE GENERATED STATEMENTS WOULD BE ENCLOSED IN BOXES ON THE OUT-  */
/*  PUT.                                                             */
/*                                                                   */
/*  EXPRESSIONS MAY BE PREDEFINED AND MINOR OPCODES MAY ALSO BE      */
/*  PREDEFINED FOR REPEATED USAGE (SEE SUBROUTINE REPLAC).           */
/*                                                                   */
/*  ALL DELIMETERS HAVE A DEFAULT VALUE WHICH MAY BE CHANGED AT      */
/*  ANY TIME USING THE 'RE' OPCODE (SEE SUBROUTINE REDEF).           */
/*                                                                   */
/*  EXTENSIVE SYNTAX CHECKING IS PERFORMED AND UNLESS AN UNRECOVER-  */
/*  ABLE ERROR IS DETECTED, PROCESSING WILL CONTINUE EVEN THOUGH     */
/*  ERRORS ARE PRESENT.  THE USER IS ADVISED TO EXAMINE THE OUTPUT   */
/*  OF THE PROGRAM CAREFULLY BEFORE INSERTING THE DATASET INTO THE   */
/*  COURSEWRITER SYSTEM.                                             */
/*                                                                   */
/*********************************************************************/
```

```
DCL (GENCAI,LOCATE,PRINT,SEPRAT,VALCHK) ENTRY,
    TRIM ENTRY (CHAR(255) VARYING) RETURNS(CHAR(255) VARYING),
    INCAI FILE INPUT STREAM,
    PRNTCAI OUTPUT PRINT STREAM,
    LETTER CHAR(1),
    MAJOR CHAR(255) VARYING,
    STR2 CHAR(255) VARYING,
    MAJOR2 CHAR(3),
    CAITAPE FILE STREAM,
    PRT BIT(1) INIT('0'B),
    TSTMTS FIXED BINARY(15,0) INIT(10B),
    NOSTMT FIXED BINARY(15,0) INIT(1B),
    (CARDS,ERROR,STMTS) FIXED BINARY(15,0) INIT(0B),
    CHANGE BIT(1),
    (TRU,TRUE) BIT(1),
    READ BIT(1) INIT('0'B),
    EOB CHAR(1) INIT(' '),
    CC1 CHAR(3) INIT('   '),
    CC2 CHAR(3) INIT('*¬#'),
    BAR FIXED BINARY(15,0),
    COUNT FIXED BINARY(15,0),
    STR CHAR(255) VARYING;
DCL DEL(11) CHAR(1) INIT('(',')','<','>','$','<','|',':','*',
                         '¬','#'),
    SET BIT(1) INIT ('0'B),
    N CHAR(8),
    J DEC FIXED(5,0),
    OPCODE(15) CHAR(3) INIT('CA ','WA ','UN ','AA ','AB ',
                            'CB ','WB ','CA(','WA(','AA(',
                            'AB(','CB(','WB(','PR ','PR('),
    EXPAND(0:31) CHAR(255) VARYING,
    PMIN(0:31) CHAR(255) VARYING,
    PMAE(0:31) CHAR(3) VARYING,
    (REPLAC,REDEF) ENTRY;
DCL PAGES FIXED DEC(4,0) INIT(0),
    HR DEC FIXED (2,0),
    SCD CHAR(7),
    D CHAR(6) DEF DAT,
    T CHAR(9) DEF TIM,
    1 DAT,
      2 YEAR CHAR(2),
      2 MONTH CHAR(2),
      2 DAY CHAR(2),
    1 TIM,
      2 HOUR CHAR(2),
      2 MIN CHAR(2),
      2 SEC CHAR(2),
      2 MSEC CHAR(3);
DCL (EXCP,EXCP3,DIF) FIXED DEC(4,0);
```

```
ON ENDFILE(INCAI) BEGIN;
                        CALL PRTERR(16,00);
                        TSTMTS=TSTMTS+1B;
                        MAJOR='END';
                        MAJOR2='END';
                  END;

ON ENDPAGE(PRNTCAI) BEGIN;
                  PAGES=PAGES+1;
                  PUT FILE(PRNTCAI) EDIT
                      ('COURSEWRITER III PREPROCESSOR',
                       'VERSION AVS.76.183','DATE: ',MONTH,
                       '/',DAY,'/',YEAR,'TIME: ',HR,':',
                       MIN,':',SCD,'PAGE ',PAGES,
                       '_____ ___ _____',(132)' ')
                      (PAGE,A,X(10),A,X(9),A,A(2),A,A(2),A,
                       A(2),X(9),A,P'99',A,A(2),A,A(7),X(10),A,
                       P'9999',SKIP(0),A,SKIP(2),A);
            END;
```

```
      OPEN FILE(INCAI),FILE(PRNTCAI) LINESIZE(132),FILE(CAITAPE)
                                                          OUTPUT;
/*  GET DATE AND EDIT TIME FOR OUTPUT HEADER                      */
      D=DATE;
      T=TIME;
      HR=DECIMAL(HOUR,2,0);
      IF HOUR>'11'
         THEN DO;
                 IF HOUR>'12' THEN HR=HR-12;
                 SCD=SEC||' P.M.';
              END;
         ELSE DO;
                 SCD=SEC||' A.M.';
                 IF HOUR='00' THEN HR=12;
              END;
      PUT FILE(PRNTCAI) EDIT ((132)' ') (LINE(60),A);
      PUT FILE(CAITAPE) EDIT ('SIGN ON'||E08,NOSTMT)
                             (A,X(64),P'(8)9');
      PUT FILE(PRNTCAI) EDIT ('*SIGN ON',NOSTMT)
                             (SKIP,A,COL(120),P'(8)9');
```

```
/*****************************************************************************/
/*                                                                           */
/*  THIS IS THE MAIN ROUTINE THAT READS EACH CARD SEQUENTIALLY AND           */
/*  DETERMINES IN WHICH OF FOUR CATEGORIES IT BELONGS.                       */
/*                                                                           */
/*  FIRST, THE DATA COULD BE ONE OF THE 15 OPCODES THAT WILL INVOKE          */
/*  THE TEXT GENERATION ROUTINES.  THIS IS DESCRIBED IN SECTION 1.           */
/*                                                                           */
/*  SECCND, THE DATA COULD BE A 'RE' OPCODE SPECIFYING THAT ONE OR           */
/*  MORE OF THE 11 DELIMETERS ARE TO BE REDEFINED.  THIS IS DES-             */
/*  CRIBED IN SECTION 2.                                                     */
/*                                                                           */
/*  THIRD, THE DATA COULD BE ON OF TWO TYPES OF PREDEFINITION CARDS          */
/*  THE FIRST ONE IS PREDEFINED MAJOR EXPRESSION WHICH IS IDENTIFIED         */
/*  BY A '$' (DEFAULT) IN COLUMN 1 AND THEN A NUMBER (0 TO 31 INC-           */
/*  LUSIVE).  SECTION 3 DESCIBES THE ACTION TAKEN IF THIS IS THE             */
/*  THE CASE.                                                                */
/*  THE SECOND PREDEFINITION TYPE, DESCRIBED IN SECTION 4, IS THE            */
/*  PREDEFINED MINOR OPCODE DENOTED BY A '<' (DEFAULT) IN COLUMN 1           */
/*  FOLLCWED BY A NUMBER (0 TO 31).                                          */
/*                                                                           */
/*  IF THE DATA FITS NONE OF THE ABOVE, THEN IT IS PASSED ON TO THE          */
/*  OUTPUT FILE WITH AN EOB CHARACTER AND A SEQUENCE NUMBER ADDED            */
/*  TO IT.                                                                   */
/*                                                                           */
/*  A CARD WITH 'END' IN CC 1 - 3 TERMINATES THE READ PORTION OF             */
/*  THE PROGRAM AND AND A RELISTING FOLLOWS.                                 */
/*---------------------------------------------------------------------------*/
/*  VARIABLES USED:                                                          */
/*      MAJOR - OPCODE READ FROM THE INPUT CARD (CC 1-3)                     */
/*      LETTER - CHARACTER EXAMINED TO SEE WHAT ACTION TO TAKE               */
/*      PRT - BIT SWITCH TO DETECT WHETHER A STATEMENT HAS ALREADY           */
/*            BEEN PRINTED OR NOT                                            */
/*      READ - BIT SWITCH TO DETERMINE WHETHER TO READ NEXT CARD             */
/*             BEFORE CONTINUING PROCESSING                                  */
/*      CAITAPE - OUTPUT FILE THAT CONTAINS PROCESSED PROGRAM                */
/*      PRNTCAI - OUTPUT FILE THAT LISTS BOTH THE INPUT AND OUTPUT           */
/*                DATASETS                                                   */
/*      INCAI - INPUT FILE                                                   */
/*      STR - THE INPUT STRING READ FROM THE CARD                            */
/*      DEL - ARRAY THAT CONTAINS THE DELIMETERS FOR THE INPUT DATA          */
/*            INITIALIZED SO THAT EVERYTHING HAS A DEFAULT                   */
/*      SET - BIT SWITCH THAT IS SET TO TRUE IF THERE HAS BEEN A             */
/*            PME DETECTED PREVIOUSLY                                        */
/*      OPCODE - ARRAY THAT CONTAINS THE 15 POSSIBLE OPCODES THAT            */
/*               CAN INVOKE THE GENCAI ROUTINE                               */
/*      EXPAND - ARRAY THAT CONTAINS THE TEXT ASSOCIATED WITH THE            */
/*               PME SYMBOL CONTAINED IN THE PMAE ARRAY                      */
/*      PMAE - ARRAY THAT IS A SYMBOL TABLE THAT HOLDS THE ASSOCI-           */
/*             ATED TEXT FOR THE PREDEFINED MAJOR EXPRESSION                 */
/*      PMIN - ARRAY THAT CONTAINS THE OPCODE AND TEXT FOR A PMO             */
/*      TSTMTS - TOTAL STEMENTS GENERATED                                    */
/*      NOSTMTS - TOTAL STATEMENTS IN PROGRAM                                */
/*                                                                           */
/*****************************************************************************/
```

```
/*****************************************************************************/
/*                          -- SECTION 1 --                                  */
/*                                                                           */
/*  AFTER THE CARD(S) IS READ AND PRINTED, THIS SECTION IS EXECUTED          */
/*  TO DETERMINE WHETHER OR NOT THE INPUT OPCODE IS ONE OF THE 15            */
/*  THAT WILL INVOKE THE GENCAI ROUTINE.  A LOOP IS EXECUTED FROM            */
/*  1 TO 15 AND COMPARISONS ARE MADE WITH THE OPCODE ARRAY.  IF A            */
/*  MATCH IS FOUND, THEN READNEXT IS CALLED TO SEE IF THERE ARE ANY          */
/*  CONTINUATIONS OF THE STATEMENT.  NEXT, THE LENGTH OF THE INPUT           */
/*  STRING IS FOUND.  IF THERE WAS A PME IN THE PREVIOUS DATA, THEN          */
/*  SET IS TRUE AND ROUTINE REPLAC IS CALLED TO SEE IF ANY STRING            */
/*  SUBSTITUTIONS SHOULD BE MADE.  ORCHK IS THEN CALLED TO SEE IF            */
/*  THERE SHCULD BE ANY CODE GENERATED.  IF THERE WAS, THEN PRT WILL         */
/*  BE ON (STATEMENTS WERE ALREADY PRINTED), OTHERWISE THE STATE-            */
/*  MENT IS PRINTED AT THIS TIME IN BOTH OUTPUT FILES.                       */
/*                                                                           */
/*  AFTER THIS SEQUENCE HAS BEEN EXECUTED, PROVIDING NO END CARD             */
/*  HAS ENCOUNTERED, VARIABLES ARE SET TO THE VALUES OBTAINED FROM           */
/*  THE LAST CARD READ SO THAT NEW CHECKS MAY BE MADE UPON IT.  THE          */
/*  DO PARAMTER IS SET TO 0 AND THE COMPARING SEQUENCE IS RE-EXEC-           */
/*  UTED.                                                                    */
/*  IF AN END CARD WAS FOUND, THEN MAJOR IS SET TO END WHICH WILL            */
/*  TERMINATE THE READ LOOP.  TRU IS SET TO ON SO THAT NO PRINTING           */
/*  WILL TAKE PLACE AT THIS PARTICULAR TIME.                                 */
/*                                                                           */
/*  IF NO MATCH IS FOUND, THEN SECTION 2 IS EXECUTED.                        */
/*****************************************************************************/

      DO WHILE (MAJOR¬='END');
         IF ¬READ
            THEN DO;
                    GET FILE(INCAI) EDIT (MAJOR,STR)
                                         (COL(1),A(3),A(68));
                    CARDS=CARDS+1;
                    STR=TRIM(STR);
                    STR=TRANSLATE(STR,CC1,CC2);
                 END;
         READ='0'B;
         K=3;
         IF MAJOR='   ' THEN K=1;
         IF MAJOR¬='END' THEN PUT FILE(PRNTCAI) EDIT (MAJOR,STR)
                                                (SKIP(K),A(3),A(68));
         LEN=LENGTH(STR);
         TRU='0'B;
         DO I=1 TO 15 WHILE (TRU='0'B);
            IF MAJOR=OPCODE(I)
               THEN DO;
                       CALL READNEXT;
                       LEN=LENGTH(STR);
                       IF SET THEN CALL REPLAC;
                       CALL ORCHK;
```

```
                IF ¬PRT
                   THEN DO;
                           NOSTMT=NOSTMT+1B;
                           PUT FILE(PRNTCAI) EDIT (NOSTMT)
                                                  (COL(120),P'(8)9');
                           STR=STR||EOB;
                           PUT FILE(CAITAPE) EDIT (MAJOR,STR,
                                                   NOSTMT)
                               (A(3),A(69),P'(8)9');
                           PRT='1'B;
                        END;
                IF MAJOR2¬='END'
                   THEN DO;
                           MAJOR=MAJOR2;
                           STR=STR2;
                           I=0B;
                           PRT='0'B;
                           TRU='0'B;
                           PUT FILE(PRNTCAI) EDIT (MAJOR,STR)
                                                  (SKIP(3),A(3),A(68));
                           LEN=LENGTH(STR);
                        END;
                   ELSE DO;
                           MAJOR='END';
                           TRU='1'B;
                        END;
             END;
END;
```

```
/*****************************************************************************/
/*                          -- SECTION 2 --                                  */
/*                                                                           */
/*  THIS SECTION IS EXECUTED WHEN THE OPCODE DOES NOT MATCH ONE OF           */
/*  THE 15 (THIS IS DETECTED BY TRU BEING SET TO OFF).  THIS MEANS           */
/*  THAT THE OPCODE MIGHT BE A SPECIAL OPCODE (RE, PME, PMO).  IF            */
/*  IN FACT IT IS AN RE, THEN ACTION IS TAKEN IN THIS SECTION.  IF           */
/*  MAJOR IS RE THEN THE REDEF ROUTINE IS CALLED WHERE THE DELI-             */
/*  METER REDEFINITION TAKES PLACE.  THE STATEMENT WILL BE PRINTED           */
/*  LATER, BUT NO SEQUENCE NUMBER WILL FOLLOW IT SINCE IT IS NOT A           */
/*  VALID COURSEWRITER III STATEMENT.  IF THIS SECTION IS EXECUTED,          */
/*  THEN THE LOOP RESTARTS WITH SECTION 1, OTHERWISE IT CONTINUES            */
/*  WITH EITHER SECTION 3 OR 4.                                              */
/*****************************************************************************/
          IF ¬TRU
             THEN DO;
                     IF MAJOR='RE '
                        THEN CALL REDEF;
```

```
/*****************************************************************************/
/*                          -- SECTION 3 --                                  */
/*                                                                           */
/*  IF NIETHER SECTIONS 1 NOR 2 ARE EXECUTED, THEN IT IS POSSIBLE            */
/*  THAT SECTION 3 WILL BE.  THIS IS THE SECTION THAT DETERMINES IF          */
/*  THE OPCODE IS IN THE FORMAT FOR A PREDEFINED MAJOR EXPRESSION            */
/*  (PME).  THESE ARE USED FOR DEFINING SEVERAL LINES OF CODE OR             */
/*  PHRASES THAT MIGHT BE USED REPEATEDLY WITH A SINGLE SYMBOL, IN           */
/*  THIS CASE A '$'.  UP TO 255 CHARACTERS CAN BE STORED USING THE           */
/*  PME.                                                                     */
/*                                                                           */
/*  THE FORMAT WILL BE A $NUMBER SO LETTER CONTAINS THE FIRST CHAR-          */
/*  ACTER OF THE OPCODE TO CHECK.  IF THIS LETTER IS A '$' (MAY BE           */
/*  OVERRIDDEN, THAT'S WHY CHECKING ARRAY LOCATION 5) THEN THE FOL-          */
/*  LOWING SEQUENCE IS PERFORMED.                                            */
/*                                                                           */
/*  FIRST, PRT IS SET TO ON BECAUSE ROUTINE READNEXT WILL PRINT THE          */
/*  STATEMENT WHEN IT IS CALLED.  AFTER THE STATEMENT IS COMPLETELY          */
/*  READ BY READNEXT, THE NEXT TWO CHARACTERS IN THE OPCODE ARE              */
/*  COMPARED TO THE CHARACTER FORM OF A NUMBER FROM 0 TO 31.  IF             */
/*  THERE IS A MATCH (THE TWO CHARACTERS REPRESENT A VALID NUMBER),          */
/*  ITS SYMBOL IS STORED IN THE PMAE ARRAY AND THE TEXT IS STORED            */
/*  IN THE EXPAND ARRAY.  LATER, WHEN THE SYMBOL IS USED, REPLAC             */
/*  WILL TAKE THE TEXT IN EXPAND AND SUBSTITUTE IT.  SET IS NOW ON           */
/*  BECAUSE THERE HAS BEEN A PME ENCOUNTERED.  THE INPUT TEXT MAY            */
/*  ALSO CONTAIN PME'S AS LONG AS THERE IS NO CIRCULAR DEFINITION.           */
/*                                                                           */
/*  IF CONVRT RETURNS A 0 IN TRU, THEN IT COULD NOT CONVERT THE              */
/*  OPCODE TO A NUMBER AND THE ERROR ROUTINE IS CALLED THAT PRINTS           */
/*  ERROR 14 (CONVERSION).                                                   */
/*                                                                           */
/*  THE VARIABLES IN EITHER CASE ARE THEN SET TO THE CONTENTS OF             */
/*  THE LAST CARD READ SO CHECKS MAY BE MADE UPON IT.                        */
/*****************************************************************************/

                    ELSE DO;
                            LETTER=SUBSTR(MAJOR,18,18);
                            IF LETTER=DEL(5)
                               THEN DO;
                                       PRT='1'B;
                                       CALL READNEXT;
                                       CALL CONVRT;
                                       IF TRU
                                          THEN DO;
                                                  MAJOR=TRIM(MAJOR);
                                                  PMAE(J)=MAJOR;
                                                  EXPAND(J)=STR;
                                                  SET='1'B;
                                                END;
                                          ELSE CALL PRTERR(14,00);
                                       MAJOR=MAJOR2;
                                       STR=STR2;
                                       PRT='1'B;
                                       READ='1'B;
                                    END;
```

```
/*****************************************************************/
/*                        -- SECTION 4 --                        */
/*                                                               */
/*  IF THIS SECTION IS EXECUTED, THEN THE OPCODE IS PREDEFINED MIN-  */
/*  OR OPCODE (PMO).  THIS ALLOWS THE USER TO DEFINE A MINOR OPCODE  */
/*  AND TEXT IN TERMS OF A NUMBER (0 TO 31).  THIS IS BETTER THAN    */
/*  USING A PME BECUASE ITS SUBSTITUTION IS DONE AT PRINT TIME RAT-  */
/*  HER THAN BEFORE ANY FURTHER PROCESSING CONTINUES.  HOWEVER, THE  */
/*  MAIN DIFFERENCE IS THAT PMO'S ARE USED TO DEFINE ONE AND ONLY    */
/*  ONE MINOR OPCODE,  ITS FORMAT IS '<' (ALSO MAY BE OVERRIDDEN)    */
/*  FOLLOWED BY A NUMBER.  MOSTLY, THE SAME PROCDURE AS FOR SECTION  */
/*  3 IS EXECUTED IF CONVRT RETURNS A 1 IN TRU.  PMIN RETAINS BOTH   */
/*  THE OPCODE AND TEXT SPECIFIED BY THE PMO.  IF CONVRT RETURNS A   */
/*  FALSE VALUE THEN ERROR MESSAGE 15 IS PRINTED FOR AN INVALID      */
/*  CONVERSION.                                                      */
/*****************************************************************/

                                ELSE IF LETTER=DEL(6)
                                     THEN DO;
                                             PRT='1'B;
                                             CALL READNEXT;
                                             CALL CONVRT;
                                             IF TRU
                                               THEN PMIN(J)=STR;
                                               ELSE CALL PRTERR
                                                         (15,00);
                                             MAJOR=MAJOR2;
                                             STR=STR2;
                                             READ='1'B;
                                             PRT='1'B;
                                          END;
                             END;
                    END;
```

```
/*************************************************************************/
/*  IF A STATEMENT HAS NOT BEEN PRINTED YET, THEN PRT WILL BE OFF.       */
/*  THIS SECTION PRINTS THE STATEMENT IN BOTH OUTPUT FILES.              */
/*  ALSO, IF NONE OF THE ABOVE SECTIONS FOUND A MATCH THEN THE           */
/*  STATEMENT IS PASSED TO THE OUTPUT FILES AT THIS POINT AND THE        */
/*  LOOP BEGINS AGAIN.                                                   */
/*************************************************************************/

          IF ¬PRT
             THEN IF MAJOR¬='END'
                     THEN DO;
                             NOSTMT=NOSTMT+1B;
                             PUT FILE(PRNTCAI) EDIT (NOSTMT)
                                                 (COL(120),P'(8)9');
                             STR=STR||EOB;
                             PUT FILE(CAITAPE) EDIT (MAJOR,STR,NOSTMT)
                                                  (A(3),A(69),P'(8)9');
                          END;
             PRT='0'B;
          END;

/*************************************************************************/
/*  ONCE AN 'END' CARD IS FOUND, THE ABOVE LOOP TERMINATES AND THE       */
/*  END STATEMENT IS ADDED TO THE FILE.  A SIGN OFF STATEMENT IS         */
/*  CREATED BECAUSE IT IS REQUIRED BY AUTO INSERT.  ALSO REQUIRED        */
/*  BY AUTO INSERT IS THAT THE LAST BLOCK OF OUTPUT BE FULL EVEN IF      */
/*  IT IS ONLY BLANKS.  THEREFORE A CHECK IS MADE TO SEE THAT A          */
/*  MULTIPLE OF 3 EXCP'S HAS BEEN PERFORMED AND IF NOT A MULTIPLE        */
/*  OF 3, THE DIFFERENCE IS ACCOUNTED FOR.                               */
/*                                                                       */
/*  STATISTICS ARE OUTPUT GIVING THE TOTAL CARDS READ, TOTAL STATE-      */
/*  MENTS GENERATED, TOTAL STATEMENTS IN THE PROGRAM, AND THE TOTAL      */
/*  NUMBER OF ERRORS THAT WERE ENCOUNTERED DURING EXECUTION.             */
/*************************************************************************/

          NOSTMT=NOSTMT+1;
          PUT FILE(CAITAPE) EDIT ('END'||EOB,NOSTMT)
                                 (A,X(68),P'(8)9');
          PUT FILE(PRNTCAI) EDIT ('END',NOSTMT)
                                 (SKIP(3),A,COL(120),P'(8)9');
          NOSTMT=NOSTMT+1B;
          PUT FILE(CAITAPE) EDIT ('SIGN OFF'||EOB,NOSTMT)
                                 (A,X(63),P'(8)9');
          PUT FILE(PRNTCAI) EDIT ('*SIGN OFF',NOSTMT)
                                 (SKIP(3),A,COL(120),P'(8)9');
          EXCP3=FIXED(NOSTMT,4,0);
          EXCP=DIVIDE(EXCP3,0003,4,0);
          EXCP=MULTIPLY(EXCP,0003,4,0);
          DIF=0003-(EXCP3-EXCP);
          IF DIF¬=3
             THEN DO J=1 TO DIF;
                     PUT FILE(CAITAPE) EDIT ((80)' ') (A);
                  END;
```

```
      PUT FILE(PRNTCAI) EDIT
                        ((132)' ',
                         'COURSEWRITER III PREPROCESSOR SUMMARY ST'||
                         'ATISTICS',
                         'NUMBER OF CARDS READ:  ',CARDS,
                         'NUMBER OF STATEMENTS GENERATED:  ',TSTMTS,
                         'NUMBER OF STATEMENTS IN PROGRAM:  ',NOSTMT,
                         'NUMBER OF ERRORS:  ',ERROR,
                         (132)' ')
                        (LINE(60),A,SKIP(2),A,SKIP(3),A,COL(44),
                         F(5,0),SKIP(2),A,COL(44),F(5,0),SKIP(2),
                         A,COL(44),F(5,0),SKIP(2),A,COL(44),F(5,0),
                         LINE(60),A);
      CLOSE FILE(INCAI),FILE(CAITAPE);

/*********************************************************************/
/*  THIS FINAL SECTION DOES A SIMPLE READ TO PRINT FOR THE ENTIRE    */
/*  CAITAPE FILE TO SHOW THE USER EXACTLY WHAT THE COURSEWRITER      */
/*  PROGRAM IS GOING TO LOOK LIKE.  IT IS A LITTLE MORE READABLE     */
/*  THAN THE FIRST LISTING BECAUSE IT HAS ONLY THE COURSEWRITER      */
/*  TEXT LISTED.  WHEN 'SIGN OFF' IS FOUND, THE PROGRAM TEMINATES.   */
/*********************************************************************/

      PUT FILE(PRNTCAI) EDIT ('THE FOLLOWING IS THE PREPROCESSED C'||
                              'OURSEWRITER III PROGRAM',
                              '___ ________ __ ___ ___________ _'||
                              '__________ ___ _______',' ')
                             (SKIP(2),X(38),A,SKIP(0),X(38),A,
                              SKIP(2),A);
      OPEN FILE(CAITAPE) INPUT;
      STR='SIGN OFF';
      DO WHILE (MAJOR¬=STR);
         GET FILE(CAITAPE) EDIT (MAJOR,STR2) (A(8),A(72));
         PUT FILE(PRNTCAI) EDIT (MAJOR,STR2) (SKIP,X(26),A(8),A(72));
      END;
      CLOSE FILE(CAITAPE),FILE(PRNTCAI);
```

```
CONVRT: PROC;


/*************************************************************************/
/*                                                                       */
/*  THIS ROUTINE TAKES AN INPUT STRING BEGINNING WITH EITHER A '$'       */
/*  OR '<' AND ATTEMPTS TO CONVERT THE REST OF THE STRING TO A NUM-      */
/*  BER IN THE RANGE OF 0 TO 31.  IT MUST MATCH A CHARACTER STRING       */
/*  WITH A CHARACTER REPRESENTATION OF A NUMBER BECUASE OF THE POS-       */
/*  SIBILITY OF INVALID INPUT.                                           */
/*                                                                       */
/*  THE METHOD USED IS TO BREAK THE STRING INTO 2 PARTS.  THE FIRST      */
/*  CHARACTER SHOULD ALWAYS BE A NUMBER SO THEN LOOK AT THE SECOND       */
/*  CHARACTER.  IF IT IS BLANK, THEN THE CHARACTER STRING IS THE         */
/*  REPRESENTATION OF A NUMBER LESS THAN 10.  IF NOT, THEN THE NUM-      */
/*  BER IS GREATER THAN 9.  SEVEN BLANKS ARE CONCATENATED TO THE         */
/*  FRONT OF THE STRING IF THE NUMBER IS LESS THAN 10, SIX BLANKS        */
/*  OTHERWISE.  THIS IS BECAUSE PL/I CONVERTS NUMBERS TO CHARACTERS      */
/*  IN SUCH A WAY THAT LEADING BLANKS ARE ADDED.  AFTER THIS, LOOP       */
/*  FROM 0 TO 31 LOOKING FOR A MATCH OF THE CHARACTER STRING TO THE      */
/*  CHARACTER FORM OF THE DO PARAMETER.  IF A MATCH IS FOUND, THEN       */
/*  TRU IS SET TO '1' INDICATING THAT THE INPUT WAS VALID.  IF NO        */
/*  MATCH COULD BE MADE, THEN THERE IS AN ERROR AND TRU IS SET TO        */
/*  FALSE WHERE AN ERROR WILL BE PRINTED IN THE MAIN PROGRAM.            */
/*-----------------------------------------------------------------------*/
/*  VARIABLES USED:                                                      */
/*       NUM - FIRST CHARACTER OF INPUT STRING                           */
/*       LETTER - SECOND CHARACTER OF INPUT STRING                       */
/*       J - DO PARAMETER                                                */
/*       N - CHARACTER REPRESENTATION OF THE DO PARAMETER                 */
/*       TEM - 0 CHARACTER REPRESENTATION OF THE 2 CHARACTER INPUT        */
/*       TRU - SWITCH TO INDICATE SUCCESSFUL RETURN                      */
/*                                                                       */
/*************************************************************************/

        DCL NUM CHAR(1),
            TEM CHAR(8);

        NUM=SUBSTR(MAJOR,2,1);
        LETTER=SUBSTR(MAJOR,3,1);
        IF LETTER=' ' THEN TEM=(7)' '||NUM;
                      ELSE TEM=(6)' '||NUM||LETTER;
        DO J=0 TO 31;
           N=CHAR(J);
           IF N=TEM THEN DO;
                            TRU='1'B;
                            RETURN;
                         END;
        END;
        END CONVRT;
```

```
GENCAI: PROC(TEXT) RECURSIVE;


/*************************************************************************/
/*                                                                       */
/*  THIS ROUTINE IS THE RECURSIVE SUBPROGRAM THAT FORMS THE VALID        */
/*  COURSEWRITER III STRINGS FROM THE CONDENSED FORM OF THE INPUT.       */
/*  THE STRING MAY BE A MAXIMUM OF 255 CHARACTERS.                       */
/*                                                                       */
/*  THE PROCEDURE USED IS TO FIRST LOCATE THE FIRST (AND POSSIBLY        */
/*  ONLY) OUTERMOST 'OR' EXPRESSION.  NEXT, RETURN THE FIRST EXP-        */
/*  RESSION FOUND WITHIN BALANCED PARENTHESES (DEFAULT).  VARIABLES      */
/*  LEN1 AND LEN2 ARE ARBITRARILY SET TO 0 AND 15 IN ORDER TO BEGIN      */
/*  A LOOP THAT PROCESSES AS LONG AS LEN1 IS LESS THAN LEN2.  THE        */
/*  LOOP, THEN, IS AS FOLLOWS:                                           */
/*    1.  SET THE RECURSION INDICATOR FOR ROUTINE SEPRAT                 */
/*    2.  INITIALIZE THE VARIABLE THAT CONTAINS THE INDIVIDUAL           */
/*        STRING WITHIN THE 'OR' EXPRESSION                              */
/*    3.  CREATE A SENTENCE BY CONCATENATING EVERYTHING UP TO THE        */
/*        SEQUENCE OF 'OR'-S, THE RETURNED STRING, AND EVERYTHING        */
/*        FOLLOWING THE EXPRESSION (ANY OF WHICH MAY BE NULL)            */
/*    4.  CHECK TO SEE IF THERE IS ANOTHER 'OR' IN THE NEW SENTENCE      */
/*    5.  IF THERE ARE NO MORE 'OR'-S IN THE SENTENCE, THEN CALL         */
/*        THE PRINT ROUTINE.  IF THERE ARE MORE 'OR'-S, THEN RECURSE     */
/*        REPEATING STEPS 1-5 UNTIL THERE ARE NO MORE EXPRESSIONS        */
/*        IN THE STRING                                                  */
/*    6.  SET LEN2 TO BE THE LENGTH OF THE RETURNED EXPRESSION           */
/*    7.  SET LEN1 TO BE THE LENGTH OF THE INDIVIDUAL WORD RETURNED      */
/*        BY SEPRAT AND ADD 2 (SINCE THE STRING HAS AN EXTRA 'OR'        */
/*        SYMBOL USED AS AN ENDING MARKER)                               */
/*    8.  IF THE TWO LENGTHS ARE EQUAL THEN ADVANCING DOWN THE EXP-      */
/*        RESSION IS COMPLETE EXCEPT FOR A NULL STRING THAT APPEARS      */
/*        AT THE END OF THE EXPRESSION.  SO ADD 1 TO LEN2 AND RE-        */
/*        PEAT STEPS 1-7                                                 */
/*    9.  IF LEN1 IS LESS THAN LEN2, THEN ELIMINATE THE WORD RETURN-     */
/*        ED BY SEPRAT FROM THE 'OR'-ED EXPRESSION AND GET THE NEXT      */
/*        WORD IN THE INPUT STRING AND REPEAT STEPS 1-9                  */
/*                                                                       */
/*************************************************************************/

      DCL (TEXT,PART,WORD,SNTNCE,REST,BEGIN) CHAR(255) VARYING,
          IND BIT(1),
          RPAREN FIXED BINARY(15,0);

      CALL LOCATE(TEXT,DEL(1),DEL(2),LPAREN,RPAREN,BEGIN,REST);
      PART=SUBSTR(TEXT,LPAREN+1B,RPAREN-1B)||DEL(7);
      LEN1=0B;
      LEN2=1111B;
```

```
DO WHILE (LEN1<LEN2);
   IND='1'B;
   WORD='';
   CALL SEPRAT(IND,PART,WORD);
   SNTNCE=BEGIN||WORD||REST;
   BAR=INDEX(SNTNCE,DEL(7));
   IF BAR=0 THEN CALL PRINT(SNTNCE);
            ELSE CALL GENCAI(SNTNCE);
   LEN2=LENGTH(PART);
   LEN1=LENGTH(WORD)+10B;
   IF LEN1=LEN2 THEN LEN2=LEN2+1B;
   IF LEN1<LEN2 THEN PART=SUBSTR(PART,LEN1);
END;
END GENCAI;
```

```
LOCATE: PROC(TEXT,LDEL,RDEL,LEFT,RIGHT,BEGIN,REST);


/*************************************************************************/
/*                                                                       */
/*  THIS ROUTINE CALLED BY SEPRAT, GENCAI, AND PRINT LOCATES THE         */
/*  POSITION OF THE PARENTHESES AND BRACKETS IN THE INPUT STRING.        */
/*  IT IS GIVEN A STRING AND THEN EXAMINES FROM LEFT TO RIGHT LOOK-      */
/*  ING FOR A SPECIFIC DELIMETER.  IT THEN RETURNS THE CHARACTER         */
/*  STRING UP TO THE LOCATION OF THE LEFT DELIMETER, THE LOCATION        */
/*  OF THE LEFT DELIMETER, THE LOCATION OF THE RIGHT DELIMETER, AND      */
/*  THE REST OF THE INPUT STRING.                                        */
/*                                                                       */
/*  THE METHOD USED IS TO FIND THE LOCATION OF THE LEFTMOST DELI-        */
/*  METER AND THEN USE A COUNTER TO COUNT THE NUMBER OF OCCURRENCES      */
/*  OF LEFT AND RIGHT DELIMETERS.  WHEN COUNT IS 0, THEN THE STRING      */
/*  IS SEPARATED INTO PARTS AND POSITIONS ARE RETURNED.                  */
/*-----------------------------------------------------------------------*/
/*  VARIABLES USED:                                                      */
/*      TEXT - INPUT STRING                                              */
/*      BEGIN - THE CHARACTER STRING UP TO THE LOCATION OF THE FIRST     */
/*              LEFT DELIMETER.                                          */
/*      REST - THE REST OF THE CHARACTER STRING AFTER THE LAST RIGHT     */
/*             DELIMETER                                                 */
/*      LEFT - POSITION OF THE FIRST LEFT DELIMETER                      */
/*      RIGHT - DISPLACEMENT OF THE LAST RIGHT DELIMETER FROM THE        */
/*              FIRST LEFT DELIMETER                                     */
/*      LDEL - THE SYMBOL FOR THE LEFT DELIMETER                         */
/*      RDEL - THE SYMBOL FOR THE RIGHT DELIMETER                        */
/*      LETTER - SINGLE CHARACTER TO EXAMINE FROM INPUT STRING           */
/*      COUNT - COUNTER FOR THE NUMBER OF OCCURRENCES OF EACH DELI-      */
/*              METER                                                    */
/*                                                                       */
/*************************************************************************/

      DCL (TEXT,BEGIN,REST) CHAR(255) VARYING,
          RIGHT FIXED BINARY(15,0),
          (LDEL,RDEL) CHAR(1);

      COUNT=1B;
      LEFT=INDEX(TEXT,LDEL);
      RIGHT=LEFT+1B;
```

```
/*************************************************************************/
/*  THIS SECTION LOOKS AT EACH CHARACTER TO DETERMINE IF IT IS A         */
/*  DELIMETER, AND IF SO, WHAT KIND.  IF IT IS A LEFT, THEN COUNT        */
/*  IS INCREMENTED BY 1.  IF IT IS A RIGHT, THEN COUNT IS DECREMENT-     */
/*  ED BY 1.  IF NOT EITHER TYPE, COUNT IS UNAFFECTED.                   */
/*  THE VARIABLE RIGHT IS INCREMENTED IN EITHER CASE SINCE WHEN          */
/*  COUNT IS FINALLY SET TO 0, RIGHT IS THE ACTUAL POSITION OF THE       */
/*  LAST RIGHT DELIMETER.  IT IS ALSO USED TO DETERMINE WHAT FOLLOWS     */
/*  THE DELIMETED EXPRESSION.                                            */
/*************************************************************************/

      DO WHILE (COUNT¬=0B);
         LETTER=SUBSTR(TEXT,RIGHT,1B);
         IF LETTER=LDEL THEN COUNT=COUNT+1B;
                        ELSE IF LETTER=RDEL THEN COUNT=COUNT-1B;
         RIGHT=RIGHT+1B;
      END;

      REST=SUBSTR(TEXT,RIGHT);
      RIGHT=RIGHT-LEFT-1B;
      BEGIN=SUBSTR(TEXT,1,LEFT-1B);
      END LOCATE;
```

```
ORCHK: PROC;


/*************************************************************************/
/*                                                                       */
/*  THIS ROUTINE IS RESPONSIBLE FOR CALLING THE APPROPRIATE GENER-       */
/*  ATION ROUTINES BASED UPON CERTAIN CRITERIA.  INITIALLY, A CHECK      */
/*  IS MADE FOR AN 'OR' SYMBOL.  IF ONE IS NOT PRESENT, A SECOND         */
/*  CHECK IS MADE FOR A '<', INDICATING THAT, ALTHOUGH THERE SHOULD      */
/*  BE NO MAJOR OPCODE GENERATION, THERE ARE MINOR OPCODES THAT          */
/*  SHOULD BE SPLIT FROM THE INPUT STRING AND PRINTED.                   */
/*                                                                       */
/*  BEFORE ANY GENERATION TAKES PLACE, THE VALCHK ROUTINE IS CALLED      */
/*  CHECKING FOR VALID CONSTRUCTIONS.  SETTING CHANGE TO ON KEEPS        */
/*  VALCHK FROM LOOKING FOR BALANCED PARENTHESES AND ONLY LOOKS FOR      */
/*  BALANCED BRACKETS.  IF VALCHK RETURNS A 1 IN TRUE, THEN PRINT        */
/*  IS CALLED THAT WILL SPLIT THE MINOR OPCODES FROM THE MAJOR OP-       */
/*  CODES.  WHEN THIS IS COMPLETE, THE SUBPROGRAM RETURNS WITH PRT       */
/*  ON BECAUSE THE STATEMENTS HAVE ALREADY BEEN PRINTED.                 */
/*                                                                       */
/*  IF AN 'OR' IS PRESENT, HOWEVER, VALCHK IS AGAIN CALLED.  TRU AND     */
/*  TRUE ARE SWITCHES USED TO INDICATE WHETHER BOTH MAJOR AND MINOR      */
/*  DELIMETERS ARE BALANCED.  IF THE MAJOR DELIMETERS ARE UNBALANCED     */
/*  IN THE STRING, THEN NO GENERATION TAKES PLACE.  IF ONLY THE MIN-     */
/*  OR DELIMETERS ARE UNBALANCED, GENERATION OCCURS BUT ONLY TO THE      */
/*  MAJOR LEVEL.  GENCAI IS CALLED AND STATISTICS ARE UPDATED.  A        */
/*  FINAL LINE OF OUTPUT GIVING THE NUMBER OF STATEMENTS CREATED IS      */
/*  PRINTED AND THE SUBPROGRAM RETURNS.                                  */
/*                                                                       */
/*************************************************************************/

      CHANGE='0'B;
      TRUE='0'B;
      BAR=INDEX(STR,DEL(7));
      IF BAR=0B
         THEN DO;
                 IF INDEX(STR,DEL(3))=0B
                    THEN RETURN;
                    ELSE DO;
                            CHANGE='1'B;
                            CALL VALCHK;
                            IF TRUE
                               THEN DO;
                                       STMTS=0B;
                                       PUT FILE(PRNTCAI) EDIT
                                          ((132)'*') (SKIP(2),A);
                                       CALL PRINT(STR);
                                       PUT FILE(PRNTCAI) EDIT
                                          ('|',' ','*'||(130)'-'||
                                           '*',' |     ',STMTS,' S'||
                                           'TATEMENTS GENERATED.',
                                           '|',(132)'*')
                                          (SKIP,A,COL(132),A,SKIP,A,
                                           SKIP,A,F(5,0),A,COL(132),
```

```
                                        A,SKIP,A);
                                    TSTMTS=TSTMTS+STMTS;
                                    PRT='1'B;
                                END;
                            RETURN;
                        END;
        END;
TRU='0'B;
CALL VALCHK;
IF ¬TRU THEN RETURN;
STMTS=0;
PRT='1'B;
PUT FILE(PRNTCAI) EDIT ((132)'*') (SKIP(2),A);
CALL GENCAI(STR);
TSTMTS=TSTMTS+STMTS;
PUT FILE(PRNTCAI) EDIT ('|','|','*'||(130)'-'||'*','|      ',
                        STMTS,' STATEMENTS GENERATED.','|',
                        (132)'*')
                       (SKIP,A,COL(132),A,SKIP,A,SKIP,A,F(5,0),
                        A,COL(132),A,SKIP,A);
END ORCHK;
```

```
PRINT:  PROC(SNTNCE);


/*********************************************************************/
/*  THIS IS THE PRINT ROUTINE CALLED BY ORCHK AND GENCAI THAT PRINTS */
/*  THE STRING GENERATED.  IT IS POSSIBLE THAT THE INPUT COULD CON-  */
/*  TAIN MINOR OPCODES AND TEXT THAT SHOULD FOLLOW THE INPUT MAJOR   */
/*  OPCODE AND TEXT.  THEREFORE, A LOOP IS EXECUTED IF TWO CONDI-    */
/*  TIONS ARE SATISFIED.  FIRST, A MINOR LEFT DELIMETER MUST BE PRE- */
/*  SENT IN THE STRING, AND, SECOND, THE MINOR DELIMETERS MUST BE    */
/*  BALANCED,  THE BIT SWITCH TRU INDICATES WHETHER OR NOT THIS COND-*/
/*  ITION IS SATISFIED.                                              */
/*                                                                   */
/*  IF BOTH CONDITIONS ARE SATISFIED, THEN THE LOOP EXAMINES THE IN- */
/*  PUT STRING REMOVING ALL MINOR OPCODES FROM THE STRING AND CREAT- */
/*  ING A NEW STRING THAT WILL BE PRINTED LATER                      */
/*                                                                   */
/*  UPON COMPLETION OF THIS LOOP, THE STRING IS TOTALLY PROCESSED    */
/*  AND READY TO BE SENT TO BOTH OUTPUT FILES.  STATISTICS ARE UP-   */
/*  DATED AND THE STRING IS PRINTED.  THE OUTPUT IS FLAGGED WITH     */
/*  AN '*' TO SHOW THAT THE STRING WAS GENERATED.                    */
/*                                                                   */
/*  IF THERE WAS A STRING OF MINOR OPCODES CREATED, ANOTHER LOOP     */
/*  SEPARATES THESE OPCODES.  IT IS POSSIBLE TO HAVE PREDEFINED MIN- */
/*  OR OPCODES AND ORDINARY COURSEWRITER CODE IN THIS STRING.  THUS,  */
/*  A LOOP MUST DETERMINE WHICH THE STRING CONTAINS AT ANY POINT.    */
/*  AFTER THIS HAS BEEN COMPLETED, THE STRING IS SEPARATED AND IS    */
/*  PRINTED.  THE LOOP CONTINUES UNTIL THE STRING IS EXHAUSTED.      */
/*  THE SUBPROGRAM THEN RETURNS TO THE CALLING PROGRAM.              */
/*                                                                   */
/*********************************************************************/

      DCL (SNTNCE,CMD,MINOR,BEGIN,REST) CHAR(255) VARYING,
          (RARROW,COLON) FIXED BINARY (15,0),
          TEMP CHAR(255) VARYING;

      LARROW=INDEX(SNTNCE,DEL(3));
      IF LARROW¬=0B
         THEN DO;
                 MINOR='';
                 IF TRUE
                    THEN DO WHILE (LARROW¬=0B);
                            CALL LOCATE(SNTNCE,DEL(3),DEL(4),LARROW,
                                        RARROW,BEGIN,REST);
                            MINOR=MINOR||DEL(8)||SUBSTR(SNTNCE,LARROW
                                                  +1B,RARROW-1B);
                            SNTNCE=BEGIN||REST;
                            LARROW=INDEX(SNTNCE,DEL(3));
                         END;
                 MINOR=MINOR||DEL(8);
              END;
         ELSE MINOR=DEL(8);
      STMTS=STMTS+1B;
      NOSTMT=NOSTMT+1B;
```

```
PUT FILE(PRNTCAI) EDIT ('|',' |',' |     *',MAJOR,SNTNCE,NOSTMT,
                        ' |')
                       (SKIP,A,COL(132),A,SKIP,A,A(3),A(68),
                        COL(120),P'(8)9',COL(132),A);
SNTNCE=SNTNCE||EOB;
PUT FILE(CAITAPE) EDIT (MAJOR,SNTNCE,NOSTMT)
                       (A(3),A(69),P'(8)9');
IF MINOR=DEL(8) THEN RETURN;
LEN1=10B;
LEN2=111B;
DO WHILE (LEN1<LEN2);
   COLON=INDEX(MINOR,DEL(8));
   CMD=SUBSTR(MINOR,1,COLON-1B);
   IF CMD='' THEN LEN1=10B;
             ELSE DO;
                     TRU='0'B;
                     LEN1=LENGTH(CMD)+10B;
                     DO J=0 TO 31 WHILE (TRU='0'B);
                        N=CHAR(J);
                        IF J<10 THEN TEMP=(7)' '||CMD;
                                ELSE TEMP=(6)' '||CMD;
                        IF TEMP=N THEN DO;
                                       CMD=PMIN(J);
                                       IF CMD=''
                                          THEN CMD='     ';
                                       TRU='1'B;
                                      END;
                     END;
                     NOSTMT=NOSTMT+1B;
                     PUT FILE(PRNTCAI) EDIT ('|     *',CMD,
                                            NOSTMT,' |')
                                            (SKIP,A,A(68),
                                             COL(120),P'(8)9',
                                             COL(132),A);
                     CMD=CMD||EOB;
                     PUT FILE(CAITAPE) EDIT (CMD,NOSTMT)
                                            (A(72),P'(8)9');
                     STMTS=STMTS+1B;
                  END;
   LEN2=LENGTH(MINOR);
   IF LEN1<LEN2 THEN MINOR=SUBSTR(MINOR,LEN1);
END;
END PRINT;
```

```
PRTERR: PROC(ERR1,ERR2);


/*************************************************************************/
/*                                                                       */
/*  THIS IS THE ERROR MESSAGE ROUTINE.  THERE ARE 12 ERRORS THAT         */
/*  HAVE BEEN TAKEN INTO ACCOUNT.  THE TWO PARAMETERS TO THE ROUTINE     */
/*  ARE THE LOCATIONS IN THE ARRAY OF THE ERROR MESSAGE TO BE PRINT-     */
/*  ED.  IF ERR2 IS 0, THEN THE MESSAGE IS ONLY 1 LINE LONG AND THE      */
/*  SUBPROGRAM RETURNS.  IF ERR2 IS NON-ZERO, THAT ARRAY LOCATION IS     */
/*  ALSO PRINTED.                                                        */
/*                                                                       */
/*************************************************************************/

        DCL (ERR1,ERR2) FIXED DEC(2,0),
            ERRMES(16) STATIC CHAR(75) INIT
('PREPROCESSOR "OR" DOES NOT APPEAR WITHIN BALANCED MAJOR DELIMETERS.',
 'STATEMENT WILL NOT BE PREPROCESSED.',
 'MISSING MAJOR DELIMETERS.  STATEMENT WILL NOT BE PREPROCESSED.',
 'THIS PREPROCESSOR DOES NOT PROVIDE FOR NESTING OF MINOR DELIMTERS.',
 'STATEMENT WILL BE PREPROCESSED TO MAJOR LEVEL ONLY.',
 'UNBALANCED MAJOR DELIMETERS.  STATEMENT WILL NOT BE PREPROCESSED.',
 'UNBALANCED MINOR DELIMEMTERS.  STATEMENT WILL BE PARTIALLY PREPROCESS
ED.',
 'MAJOR DELIMETERS ARE THE SAME.  PREPROCESS TERMINATED.',
 'FURTHER EXECUTION WILL LIKELY RESULT IN AN UNRECOVERABLE ERROR.',
 'MINOR DELIMETERS ARE THE SAME.  PREPROCESS TERMINATED.',
 'MISSING DELIMETER IN REDEFINE STATEMENT.  STATEMENT IGNORED.',
 'ATTEMPT TO PERFORM MORE THAN 255 PME REPLACEMENTS.',
 'PROBABLE CAUSE:  TWO PME''S REFERENCE EACH OTHER.',
 'INVALID PREPROCESSOR MAJOR EXPRESSION (PME) DEFINITION.  STATEMENT DE
LETED.',
 'INVALID PREPROCESSOR MINOR OPCODE (PMO) DEFINITION.  STATEMENT DELETE
D.',
 'MISSING "END" CARD.  ONE HAS BEEN GENERATED.');
        ERROR=ERROR+1;
        PUT FILE(PRNTCAI) EDIT ('ERROR --  ',ERRMES(ERR1))
                               (SKIP,A,A(90));
        IF ERR2=0 THEN RETURN;
        PUT FILE(PRNTCAI) EDIT (ERRMES(ERR2)) (SKIP,X(10),A(90));
        END PRTERR;
```

```
READNEXT: PROC;


/*************************************************************************/
/*                                                                       */
/*  THIS ROUTINE READS THE NEXT CARD OR CARDS OF THE INPUT STREAM        */
/*  TO SEE IF THEY ARE CONTINUATIONS OF THE PRECEDING CARD READ.         */
/*  IF THE OPCODE IS 3 BLANKS, THEN THE CONTENTS OF THE CARD ARE         */
/*  CONCATENATED TO THE CONTENTS OF THE PREVIOUS CARD.  IF MAJOR2        */
/*  (OPCODE) IS NON-BLANK, THEN THE ROUTINE RETURNS AND PROCESSING       */
/*  CONTINUES.                                                           */
/*-----------------------------------------------------------------------*/
/*  VARIABLES USED:                                                      */
/*      EXIT - BIT SWITCH THAT ENDS LOOP WHEN THERE IS NO CONTINU-       */
/*             ATION                                                     */
/*      CARDS - CARD INPUT COUNTER                                       */
/*      STR2 - INPUT STRING OF FOLLOWING CARDS                           */
/*      CC1 - SPECIAL CHARACTERS TO INDICATE LINE FEEDS AND BACK-        */
/*            SPACES                                                     */
/*      CC2 - HEXADECIMAL REPRESENTATION OF LINEFEEDS AND BACKSPACES     */
/*      MAJOR2 - OPCODE ON FOLLOWING CARD                                */
/*                                                                       */
/*************************************************************************/

        DCL EXIT BIT(1) INIT('0'B);

        DO WHILE (EXIT='0'B);
           GET FILE(INCAI) EDIT (MAJOR2,STR2) (COL(1),A(3),A(68));
           CARDS=CARDS+1;
           STR2=TRIM(STR2);
           STR2=TRANSLATE(STR2,CC1,CC2);
           IF MAJOR2='   '
              THEN DO;
                      PUT FILE(PRNTCAI) EDIT (MAJOR2,STR2)
                                             (SKIP,A(3),A(68));
                      STR=STR||STR2;
                   END;
              ELSE EXIT='1'B;
        END;
        END READNEXT;
```

```
REDEF:  PROC;

/*************************************************************************/
/*                                                                       */
/*  THIS ROUTINE ALLOWS THE USER TO REDEFINE ANY OF THE DELIMETERS       */
/*  USED IN THE PREPROCESS.  THE REDEFINITION IS INVOKED WHEN THE        */
/*  RE OPCODE IS ENCOUTERED IN THE INPUT STREAM.  THE SUBPROGRAM         */
/*  SEARCHES FOR ANY OF 11 KEYWORDS AND A SYMBOL IN ORDER TO MAKE        */
/*  THE NEW ASSIGNMENT.  THE KEYWORDS MAY APPEAR IN ANY ORDER AND        */
/*  ONLY THOSE THAT ARE BEING REDEFINED NEED BE SPECIFIED.               */
/*  THE KEYWORDS ARE AS FOLLOWS:                                         */
/*                                                                       */
/*      KEY ABBREV.          MEANING                          DEFAULT    */
/*         LMA=          LEFT MAJOR DELIMETER                   (        */
/*         RMA=          RIGHT MAJOR DELIMETER                  )        */
/*         LMI=          LEFT MINOR DELIMETER                   <        */
/*         RMI=          RIGHT MINOR DELIMETER                  >        */
/*         PME=          PREDEFINED MAJOR EXPRESSION            $        */
/*         PMO=          PREDEFINED MINOR OPCODE                <        */
/*         POR=          PREPROCESSOR 'OR' SYMBOL               |        */
/*         AND=          PREPROCESSOR 'AND' SYMBOL              :        */
/*         LFD=          LINE FEED CHARACTER                    ¤        */
/*         BSP=          BACKSPACE CHARACTER                    ¬        */
/*         CRT=          CARRIAGE RETURN CHARACTER              #        */
/*                                                                       */
/*  CHECKS ARE MADE TO BE SURE THAT THERE WAS AT LEAST ONE REDEFIN-      */
/*  ITION OF A KEY, A KEYWORD HAD A DELIMETER STATED AFTER THE EQUAL     */
/*  SIGN, AND THAT LEFT AND RIGHT DELIMTERS WERE NOT DEFINED TO BE       */
/*  THE SAME SYMBOL.                                                     */
/*                                                                       */
/*************************************************************************/

    DCL KEY(11) CHAR(4) STATIC INIT ('LMA=','RMA=','LMI=','RMI=',
                                     'PME=','PMO=','POR=','AND=',
                                     'LFD=','BSP=','CRT='),
        CHANGE BIT(1) INIT ('0'B);

    PRT='1'B;
    DO J=1 TO 11;
       KEYWD=INDEX(STR,KEY(J));
       IF KEYWD¬=0B
          THEN IF KEYWD+4>LEN
                  THEN CALL PRTERR(11,00);
                  ELSE DO;
                          CHANGE='1'B;
                          DEL(J)=SUBSTR(STR,KEYWD+4,1);
                  END;
    END;
```

```
IF CHANGE
   THEN IF DEL(1)=DEL(2)
           THEN DO;
                  CALL PRTERR(08,09);
                  MAJOR='END';
                END;
           ELSE IF DEL(3)=DEL(4)
                   THEN DO;
                           CALL PRTERR(10,09);
                           MAJOR='END';
                        END;
                   ELSE PUT FILE(PRNTCAI) EDIT
                            ('DELIMETERS REDEFINED: ',
                             (KEY(J),DEL(J) DO J=1 TO 11))
                            (SKIP,A,11(A(4),A(1),X(2)));
CC2=DEL(9)||DEL(10)||DEL(11);
END REDEF;
```

```
REPLAC: PROC;


/*************************************************************************/
/*                                                                       */
/*  THE REPLAC SUBROUTINE ALLOWS THE USER TO DEFINE SEVERAL LINES        */
/*  OF CODE (UP TO 255 CHARACTERS) AND DESIGNATE THAT SECTION AS A       */
/*  $ (BY DEFAULT) FOLLOWED BY A NUMBER IN THE RANGE OF 0 TO 31.         */
/*  REPLAC, UPON ENCOUNTERING A DEFINED SYMBOL, THEN SUBSTITUTES THE     */
/*  ENTIRE SECTION IN THE PLACE OF THE $NUMBER.  THE INSERTION MAY       */
/*  TAKE PLACE ANYWHERE IN THE STRING AND THE INPUT STRING MAY IT-       */
/*  SELF BE ONLY A $NUMBER.  IF MORE THAN 255 REPLACEMENTS ARE AT-       */
/*  TEMPTED, AN ERROR MESSAGE IS PRINTED AND THE ROUTINE RETURNS TO      */
/*  THE MAIN PROGRAM.                                                    */
/*                                                                       */
/*************************************************************************/

      DCL (BEGIN,REST) CHAR(255) VARYING;

      DO K=1 TO 255;
         IF INDEX(STR,DEL(5))=0
            THEN RETURN;
         IN=0;
         DO J=0 TO 31 WHILE (IN=0);
            IN=INDEX(STR,PMAE(J));
         END;
         IF IN=0
            THEN RETURN;
         J=J-1;
         LEN2=LENGTH(PMAE(J));
         BEGIN=SUBSTR(STR,1,IN-1);
         IF IN+LEN2>LEN
            THEN REST='';
            ELSE REST=SUBSTR(STR,IN+LEN2);
         STR=BEGIN||EXPAND(J)||REST;
         LEN=LENGTH(STR);
      END;
      CALL PRTERR(12,13);
      END REPLAC;
```

```
SEPRAT: PROC(IND,PART,WORD) RECURSIVE;


/*********************************************************************/
/*                                                                   */
/*  THIS ROUTINE RECURSIVELY SEPARATES THE 'OR'-ED STRING FROM THE   */
/*  REST OF THE INPUT STRING.  IT IS RECURSIVE BECAUSE NESTING OF    */
/*  THE EXPRESSIONS IS ALLOWED.                                      */
/*                                                                   */
/*  THE INPUT STRING IS GIVEN TO THIS ROUTINE WITH THE OUTERMOST     */
/*  DELIMETERS REMOVED.  THERE IS THE POSSIBLITY THAT THERE IS ANO-  */
/*  THER SET WITHIN THE STRING.  THE OBJECT IS TO RETURN THE STRING  */
/*  UP TO AN 'OR'.                                                   */
/*                                                                   */
/*  AN EXAMPLE OF WHAT WOULD BE SUPPLIED AND RETURNED THE FIRST TIME */
/*  IS THE STRING: A(I|E(X|Y))S|B                                    */
/*  EVERYTHING UP TO AND INCLUDING THE 'S' WOULD BE RETURNED.        */
/*-------------------------------------------------------------------*/
/*  VAIABLES USED:                                                   */
/*       BAR - POSITION OF THE '|' IN THE STRING                     */
/*       WRD - CHARACTER STRING UP THE '|'                           */
/*       PART - THE INPUT STRING                                     */
/*       LPAREN - LOCATION OF THE FIRST LEFT DELIMETER               */
/*       IND - RECURSION INDICATOR THAT IS TRUE IF NO RECURSION HAS  */
/*             OCCURRED                                              */
/*       WORD - CHARACTER STRING TO BE RETURNED TO CALLING PROGRAM   */
/*              WHICH WOULD BE A(I|ENX|Y))S ON THE FIRST CALL.  ON   */
/*              THE SECOND CALL, THE 'B' WOULD BE RETURNED.          */
/*       BEGIN - THE BEGINNING OF THE INPUT STRING UP TO THE NESTED  */
/*               PARENTHESIZED EXPRESSION                            */
/*       REST - THE REMAINDER OF THE STRING FOLLOWING A PARENTHES-   */
/*              IZED EXPRESSION                                      */
/*                                                                   */
/*********************************************************************/

      DCL (RPAREN,LPAREN) FIXED BINARY(15,0),
         (WORD,WRD,REST,BEGIN,PART) CHAR(255) VARYING,
         IND BIT(1);

/*  FIND FIRST '|'                                                   */
      BAR=INDEX(PART,DEL(7));
/*  SET WRD TO BE EVERYTHING UP TO THE FIRST '|'                     */
      WRD=SUBSTR(PART,1,BAR-1B);
/*  SEE IF THERE IS A LEFT DELIMETER IN THE STRING IN WRD            */
      LPAREN=INDEX(WRD,DEL(1));
      IF LPAREN=0
/* IF NOT, THEN SEE IF DONE ANY RECURSION                            */
        THEN IF IND
/*  IF NOT (IND IS '1') THEN RETURN WRD AS THE SEPARATED WORD        */
                THEN WORD=WRD;
/*  IF YES, THEN TAKE WHAT WAS ALREADY IN WORD AND ADD THE NEW STRING
    AND RETURN                                                        */
                ELSE WORD=WORD||WRD;
```

```
/*************************************************************************/
/*  IF THERE IS A LEFT DELIMETER, THEN LOCATE THE BALANCED SET, SET  */
/*  THE RECURSION INDICATOR, LENGTHEN WORD TO INCLUDE PARTS RETURN-  */
/*  ED BY LOCATE, AND CALL SEPRAT TO FIND THE REST OF THE EXPRESSION */
/*  WHICH IS STORED IN THE VARIABLE REST                             */
/*************************************************************************/

          ELSE DO;
                  CALL LOCATE(PART,DEL(1),DEL(2),LPAREN,RPAREN,
                              BEGIN,REST);
                  WORD=WORD||BEGIN||SUBSTR(PART,LPAREN,RPAREN+1B);
                  IND='0'B;
                  CALL SEPRAT(IND,REST,WORD);
/*  KEEP ON RECURSING UNTIL THERE IS NO LEFT DELIMETER IN THE STRING */
               END;
        END SEPRAT;
```

```
TRIM:   PROC(STR) RETURNS(CHAR(255) VARYING);


/*************************************************************************/
/*                                                                       */
/*  THIS ROUTINE TRIMS ALL TRAILING BLANKS FROM THE INPUT STRING.        */
/*  IT DOES THIS BY STARTING AT THE END OF THE STRING AND WORKING        */
/*  BACKWARD TO THE ACTUAL END OF THE STRING.  IT COUNTS THE NUMBER      */
/*  OF TRAILING BLANKS AND THEN TAKES THE SUBSTRING OF THE INPUT.        */
/*-----------------------------------------------------------------------*/
/*  VARIABLES USED:                                                      */
/*       STR - THE INPUT STRING TO BE TRIMMED                            */
/*       LEN - THE CURRENT LENGTH OF THE STRING                          */
/*       BLANK - COUNTER FOR THE NUMBER OF TRAILING BLANKS               */
/*       LETTER - SINGLE CHARACTER TO EXAMINE                            */
/*       L - DO PARAMETER                                                */
/*                                                                       */
/*************************************************************************/

        DCL STR CHAR(255) VARYING;
        DCL BLANK FIXED BINARY(15,0) INIT(0B);

        LEN=LENGTH(STR);
        DO L=LEN TO 1 BY -1;
           LETTER=SUBSTR(STR,L,1);
           IF LETTER=' ' THEN BLANK=BLANK+1B;
                         ELSE DO;
                                 LEN=LEN-BLANK;
                                 STR=SUBSTR(STR,1,LEN);
                                 RETURN(STR);
                              END;
        END;
        STR='';
        RETURN(STR);
        END TRIM;
```

```
VALCHK: PROC;


/*************************************************************************/
/*                                                                       */
/*  THE PURPOSE OF VALCHK IS TO EXAMINE A STRING THAT CONTAINS CON-      */
/*  DENSED TEXT AND SHOULD BE EXPANDED FOR VALID CONSTRUCTION.  THE      */
/*  SUBPROGRAM IS BROKEN DOWN INTO TWO SECTICNS.  SECTION 1 CONCERNS     */
/*  CHECKING FOR BALANCED PARENTHESES AND ACTION TAKEN IF UNBALANCED     */
/*  PARENTHESES ARE ENCOUNTERED.  SECTION 2 IS CONCERNED WITH CHECK-     */
/*  ING FOR BALANCED BRACKETES AND ACTION TAKEN IF THEY ARE UNBAL-       */
/*  ANCED.                                                               */
/*                                                                       */
/*************************************************************************/

        DCL (MAJCNT,MINCNT) FIXED BINARY(15,0) INIT(0);

/*************************************************************************/
/*                        -- SECTION 1 --                                */
/*  THIS SECTION LOOKS AT EACH CHARACTER IN THE STRING AND DETER-        */
/*  MINES WHAT IT IS.  IT IS ONLY EXECUTED IF THE VARIABLE CHANGE IS     */
/*  SET TO OFF, MEANING THE RCUTINE SHOULD LOOK FOR UNBALANCED PAR-      */
/*  ENTHESES AROUND AN 'OR' SYMBOL.  A COUNTER IS USED THAT IS INC-      */
/*  REMENTED BY 1 FOR EVERY OCCURRENCE OF A '(', AND DECREMENTED BY      */
/*  1 FOR EACH OCCURRENCE OF A ')'.  IF MAJCNT (THE COUNTER) IS EVER     */
/*  LESS THAN 0, THEN PARENTHESES ARE UNBALANCED.  ALSO, IF MAJCNT       */
/*  IS 0, AND AN 'OR' SYMBOL (DEL 7) IS FOUND, THEN THERE IS AN ER-      */
/*  ROR.  AT THEN END OF THE LOOP (EVERY CHARACTER HAS BEEN CHECKED)     */
/*  IF THE COUNTER IS NOT 0, THEN PARENTHESES ARE UNBALANCED, AN ER-     */
/*  ROR IS PRINTED, AND THE PROGRAM RETURNS.                             */
/*************************************************************************/

        IF ¬CHANGE
           THEN DO;
                   DO J=1 TO LEN;
                      LETTER=SUBSTR(STR,J,1);
                      IF LETTER=DEL(1)
                         THEN DO;
                                 MAJCNT=MAJCNT+1B;
                                 CHANGE='1'B;
                              END;
                         ELSE IF LETTER=DEL(2)
                                 THEN DO;
                                         MAJCNT=MAJCNT-1B;
                                         IF MAJCNT<0
                                            THEN DO;
                                                    CALL PRTERR(06,00);
                                                    RETURN;
                                                 END;
                                      END;
```

```
                                ELSE IF LETTER=DEL(7)
                                     THEN IF MAJCNT=0
                                          THEN DO;
                                                  CALL PRTERR(01,02);
                                                  RETURN;
                                               END;
            END;
         END;
IF MAJCNT¬=0B
   THEN DO;
            CALL PRTERR(06,00);
            RETURN;
         END;
   ELSE IF ¬CHANGE
            THEN DO;
                    CALL PRTERR(03,00);
                    RETURN;
                 END;
            ELSE TRU='1'B;
```

```
/*****************************************************************************/
/*                          -- SECTION 2 --                                  */
/*  THIS SECTION PERFORMS APPORXIMATELY THE SAME TASK AS SECTION 1           */
/*  EXCEPT IT LOOKS FOR UNBALANCED BRACKETS AND IS EXECUTED UNCOND-          */
/*  ITIONALLY.  A COUNTER (MINCNT) IS USED TO DETERMINE IF THE BRAC-         */
/*  KETS ARE BALANCED BY ADDING OR SUBTRACTING 1.  IF MINCNT IS EVER         */
/*  GREATER THAN 1, NESTING HAS BEEN ATTEMPTED WHICH IS NOT ALLOWED          */
/*  IN THIS PROGRAM.  THE VARIABLE LOC IS SET TO THE LOCATION OF AN          */
/*  '<' SYMBOL, AND WHEN A '>' IS ENCOUNTERED, A CHECK IS MADE TO            */
/*  DETERMINE IF THE '<' OCCURS BETWEEN BRACKETS.  IF IT DOES, AN            */
/*  ERROR IS PRINTED.  AFTER EACH CHARACTER HAS BEEN CHECKED, IF             */
/*  MINCNT IS NOT 0, THERE IS AN ERROR AND TRUE IS SET TO OFF SIG-           */
/*  NIFYING THAT PROCESSING SHOULD ONLY GO TO THE MAJOR OPCODE LEVEL.*/
/*****************************************************************************/

        DO J=1 TO LEN;
           LETTER=SUBSTR(STR,J,1);
           IF LETTER=DEL(3)
              THEN DO;
                      MINCNT=MINCNT+1B;
                      IF MINCNT=2
                         THEN DO;
                                 CALL PRTERR(04,05);
                                 RETURN;
                              END;
                      LOC=J;
                   END;
              ELSE IF LETTER=DEL(4)
                      THEN DO;
                              MINCNT=MINCNT-1B;
                              IF MINCNT<0B
                                 THEN DO;
                                         CALL PRTERR(07,00);
                                         RETURN;
                                      END;
                                 ELSE DO;
                                         IF INDEX(SUBSTR(STR,LOC,J-LOC)
                                                ,DEL(7))¬=0
                                            THEN DO;
                                                    CALL PRTERR(07,00);
                                                    RETURN;
                                                 END;
                                      END;
                           END;
        END;
        IF MINCNT=0
           THEN TRUE='1'B;
           ELSE CALL PRTERR(07,00);
        END VALCHK;



END SLATE;
```

APPENDIX B

EXAMPLE USAGE OF THE SYSTEM

This section consists of two examples of SLATE instructions that a lesson author might use to implement rather complex questions in COURSEWRITER III. Following each example is the complete COURSEWRITER code that will be used by AUTO INSERT. In both examples, the first listing of the output has been deleted for cost reducing measures, and the sequencing of statements has been omitted for consistency in reduction size.

EXAMPLE 1

This example is a question on PDP-11 assembly language that was originally demonstrated using the GIZMO programming language at the University of Illinois and has now been adapted for the SLATE system. The question presented to the student is a request for the student to enter a PAL-11 instruction to add one to the system register 0. There are several correct answers, the best of which is

```
INC R0
```

The student is informed of this if he/she enters a different correct answer but one that is less efficient. The wrong answers most likely to occur are also specified.

The input stream consists of 35 statements. The output is 789 statements and the entire processing required approximately fifteen seconds of CPU time.

```
RE CRT=*
QU WRITE A PDP-11 ASSEMBLY LANGUAGE INSTRUCTION THAT ADDS THE NUMBER
   ONE TO THE CONTENTS OF REGISTER 0.
$1 (%0|R0)
<1 TY THAT'S NOT VERY EFFICIENT.
<3 TY INC R0 IS BETTER.  BUT YOURS IS O.K. TOO.
<12TY YOU'RE TRYING TO BEE A WISE GUY.
<5 TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
<2 TY VERY GOOD.
<10TY O.K. BUT A PSUEDO-OP IS NOT REALLY A PROPER PAL INSTRUCTION.
<4 TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
<30   RIGHT, NOT THE LEFT.
<19TY THE BEST ANSWER IS
<20       INC R0          ;ADD ONE TO REGISTER 0
<21   BUT   ADD#1,R0      ALSO WORKS, BUT IS LESS EFFICIENT.
<11TY IT'S A DEFAULT PSUEDO-OP
<6 TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
<8 TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
<7 TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
<13    COMMA.  YOU LEFT IT OUT.
<9 TY INCB WON'T WORK ALL THE TIME.  FOR EXAMPLE, IF LOWER BYTE OF
<14   REGISTER 0 CONTAINS 377, THEN INCB WILL NOT DO THE SAME THING
<15   AS INC.
CA ((ADD#1,$1,<1>|SUB#177777,$1<12>)<3>|(.WORD 5200|5200<11>)<10:12>|IN
   C$1<2>)
$2 ((REG0|REG|R0|0)<8>)
WA (ADDB<5>|ADD)(#1|1<6>)(,|<7:13>)$2
WA (ADDB<5>|ADD)$2(,|<7:13>)(#1|1<6>)<1:4:30>
$3 (SUBB<5>|SUB)
WA $3(#177777|177777<6>)(,|<7:13>)$2<1>
WA $3$2(,|<7:13>)(#177777|177777<6>)<1:4:30>
WA (INCB<9:14:15>|INC)$2
$5 NC, TRY AGAIN.
UN ($5|$5|$5|<19:20:21>)
END
```

THE FOLLOWING IS THE PREPROCESSED COURSEWRITER III PROGRAM

```
SIGN ON
QU WRITE A PDP-11 ASSEMBLY LANGUAGE INSTRUCTION THAT ADDS THE NUMBER
   ONE TO THE CONTENTS OF REGISTER 0.
CA ADD#1,%0,
TY THAT'S NOT VERY EFFICIENT.
TY INC R0 IS BETTER.  BUT YOURS IS O.K. TOO.
CA ADD#1,R0,
TY THAT'S NOT VERY EFFICIENT.
TY INC R0 IS BETTER.  BUT YOURS IS O.K. TOO.
CA SUB#177777,%0
TY YOU'RE TRYING TO BEE A WISE GUY.
TY INC R0 IS BETTER.  BUT YOURS IS O.K. TOO.
CA SUB#177777,R0
TY YOU'RE TRYING TO BEE A WISE GUY.
TY INC R0 IS BETTER.  BUT YOURS IS O.K. TOO.
CA .WORD 5200
TY O.K. BUT A PSUEDO-OP IS NOT REALLY A PROPER PAL INSTRUCTION.
TY YOU'RE TRYING TO BEE A WISE GUY.
CA 5200
TY IT'S A DEFAULT PSUEDO-OP
TY O.K. BUT A PSUEDC-OP IS NOT REALLY A PROPER PAL INSTRUCTION.
TY YOU'RE TRYING TO BEE A WISE GUY.
CA INC%0
TY VERY GOOD.
CA INCR0
TY VERY GOOD.
WA ADDB#1,REG0
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB#1,REG
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB#1,R0
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB#1,0
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB#1REG0
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB#1REG
TY THAT OPERATOR IS NCT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB#1R0
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB#10
```

```
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB1,REGO
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB1,REG
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB1,RO
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB1,0
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB1REGO
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB1REG
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB1RO
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDB10
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD#1,REGO
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD#1,REG
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD#1,RO
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD#1,0
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD#1REGO
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD#1REG
```

```
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD#1R0
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD#10
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD1,REG0
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD1,REG
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD1,R0
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD1,0
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD1REG0
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD1REG
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD1R0
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADD10
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA ADDBREGO,#1
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDBREGO,1
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDBREGO#1
```

```
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDBREG01
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDBREG,#1
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDBREG,1
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDBREG#1
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDBREG1
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDBRO,#1
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDBRO,1
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
```

```
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDBRO#1
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDBR01
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDB0,#1
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDB0,1
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDB0#1
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDB01
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDREG0,#1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDREG0,1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
```

```
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDREGO#1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDREGO1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDREG,#1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDREG,1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDREG#1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDREG1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDRO,#1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDRO,1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDRO#1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
```

```
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADDR01
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADD0,#1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADD0,1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADD0#1
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA ADD01
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBB#177777,REGO
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB#177777,REG
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB#177777,R0
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB#177777,0
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB#177777REGO
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
```

```
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB#177777REG
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB#177777R0
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB#1777770
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TC EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB177777,REGO
TY THAT OPERATOR IS NCT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB177777,REG
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB177777,R0
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TC EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB177777,0
TY THAT OPERATOR IS NCT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB177777REGO
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB177777REG
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB177777R0
```

```
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBB1777770
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB#177777,REGO
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB#177777,REG
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB#177777,RO
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB#177777,0
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB#177777REGO
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB#177777REG
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB#177777RO
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB#1777770
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB177777,REGO
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB177777,REG
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB177777,RO
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
```

```
WA SUB177777,0
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB177777REG0
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB177777REG
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB177777R0
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUB1777770
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
WA SUBBREGO,#177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBREGO,177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBREGO#177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBREGO177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
```

```
   RIGHT, NOT THE LEFT.
WA SUBBREG,#177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBREG,177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBREG#177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBREG177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBRO,#177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBRO,177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBRO#177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBRO177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
```

```
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBO,#177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBO,177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBO#177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBBO177777
TY THAT OPERATOR IS NOT A VALID ONE.  BETTER CHECK YOUR HANDBOOK.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBREGO,#177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBREGO,177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBREGO#177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBREGO177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
```

```
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBREG,#177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBREG,177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBREG#177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBREG177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBRO,#177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBRO,177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBRO#177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBRO177777
TY THAT IS NOT A VALID WAY TC EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUBO,#177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
```

```
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUB0,177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUB0#177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA SUB0177777
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
TY THE OPERANDS IN A 2-ADDRESS INSTRUCTION ARE SEPARATED BY A
    COMMA.  YOU LEFT IT OUT.
TY ALL CONSTANTS (LITERALS) MUST BE PREFIXED BY A # SIGN.
TY THAT'S NOT VERY EFFICIENT.
TY THIS IS NOT THE 360.  THE DESTINATION ADDRESS APPEARS ON THE
   RIGHT, NOT THE LEFT.
WA INCBREG0
TY INCB WON'T WORK ALL THE TIME.  FOR EXAMPLE, IF LOWER BYTE OF
   REGISTER 0 CONTAINS 377, THEN INCB WILL NOT DO THE SAME THING
   AS INC.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA INCBREG
TY INCB WON'T WORK ALL THE TIME.  FOR EXAMPLE, IF LOWER BYTE OF
   REGISTER 0 CONTAINS 377, THEN INCB WILL NOT DO THE SAME THING
   AS INC.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA INCBR0
TY INCB WON'T WORK ALL THE TIME.  FOR EXAMPLE, IF LOWER BYTE OF
   REGISTER 0 CONTAINS 377, THEN INCB WILL NOT DO THE SAME THING
   AS INC.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA INCB0
TY INCB WON'T WORK ALL THE TIME.  FOR EXAMPLE, IF LOWER BYTE OF
   REGISTER 0 CONTAINS 377, THEN INCB WILL NOT DO THE SAME THING
   AS INC.
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA INCREG0
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA INCREG
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA INCR0
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
WA INC0
TY THAT IS NOT A VALID WAY TO EXPRESS REGISTER 0.
UN NO, TRY AGAIN.
UN NO, TRY AGAIN.
UN NO, TRY AGAIN.
UN
TY THE BEST ANSWER IS
```

```
      INC R0          ;ADD ONE TO REGISTER 0
   BUT    ADD#1,R0       ALSO WORKS, BUT IS LESS EFFICIENT.
END
SIGN OFF
```

CRANES CREST

EXAMPLE 2

In 1975, a thesis was written by Teddy E. Sherrod which included the implementation of a CAI program to teach new operators of the System/360 at UTCC the Houston Automatic Spooling Priority (HASP) commands. This example is a possible question that may have been used in that course. A HASP command is of the form

$ verb device, line, or initiator

where the verb may be either the full word or a single letter. Assumptions for this example are that there are a maximum of six remote terminals, six lines to the remotes, seven initiators, two in-house printers and readers. The remote terminal devices are designated

RMn.PR1 (remote n printer 1)

RMn.RD1 (remote n reader 1)

In-house printers and readers are

PRT1 and PRT2

RDR1 and RDR2

Each line is designated

LNEn ($1 \leq n \leq 6$)

The initiators are denoted

In ($1 \leq n \leq 7$)

The question is to supply a HASP command to start a device, line or initiator.

The input stream is 21 statements. The output is 556 statements. Processing required approximately 18 seconds of CPU time.

```
QU GIVE A HASP COMMAND TO START A DEVICE, LINE, OR INITIATOR.
<1 TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
<2 TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
<3    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
<4 TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
<5 TY YOU STARTED THE LINE TO A REMOTE DEVICE CORRECTLY.
<6 TY THIS STARTS ALL THE INITIATORS.
<7 TY A PARTICULAR INITIATOR WILL BE STARTED.
<8 TY THE PRINTER WILL NOW BE STARTED.
<9 TY YOU STARTED THE READER CORRECTLY.
$0 (1|2|3|4|5|6
CA $S(|TART<2:3>)(RM$0).(PR1|RD1)<4>|PRT(1|2)<8>|RDR(1|2)<9>|LNE$0)<5>
   |I(<6>|$0|7)<7>)|<1>
BA(L) SG<TY YOU FORGOT THE '$' SIGN.>
<10TY WHICH REMOTE ARE YOU REFERRING TO?
<11TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
<12TY YOU FORGOT THE DEVICE NUMBER.
<13TY WHICH REMOTE LINE ARE YOU STARTING?
WA $S(|TART<2:3>)(RM$0|<10>)(<11>(PR|RD)(<12>|1)|.(PR|RD)<12>)|(PRT|RDR
   )<12>|LNE<13>)
END
```

THE FOLLOWING IS THE PREPROCESSED COURSEWRITER III PROGRAM

```
SIGN ON
QU GIVE A HASP COMMAND TO START A DEVICE, LINE, OR INITIATOR.
CA $SRM1.PR1
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SRM1.RD1
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SRM2.PR1
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SRM2.RD1
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SRM3.PR1
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SRM3.RD1
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SRM4.PR1
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SRM4.RD1
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SRM5.PR1
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SRM5.RD1
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SRM6.PR1
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SRM6.RD1
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SPRT1
TY THE PRINTER WILL NOW BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SPRT2
TY THE PRINTER WILL NOW BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SRDR1
TY YOU STARTED THE READER CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SRDR2
TY YOU STARTED THE READER CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SLNE1
TY YOU STARTED THE LINE TO A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SLNE2
```

```
TY YOU STARTED THE LINE TO A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SLNE3
TY YOU STARTED THE LINE TO A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SLNE4
TY YOU STARTED THE LINE TO A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SLNE5
TY YOU STARTED THE LINE TO A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SLNE6
TY YOU STARTED THE LINE TO A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SI
TY THIS STARTS ALL THE INITIATORS.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SI1
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SI2
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SI3
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SI4
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SI5
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SI6
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $SI7
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRM1.PR1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRM1.RD1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRM2.PR1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRM2.RD1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
```

```
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRM3.PR1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRM3.RD1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRM4.PR1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRM4.RD1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRM5.PR1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRM5.RD1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRM6.PR1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRM6.RD1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY GOOD.  YOU HAVE IDENTIFIED A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTPRT1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY THE PRINTER WILL NOW BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTPRT2
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY THE PRINTER WILL NOW BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRDR1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU STARTED THE READER CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTRDR2
```

```
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU STARTED THE READER CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTLNE1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU STARTED THE LINE TO A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTLNE2
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU STARTED THE LINE TO A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTLNE3
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU STARTED THE LINE TO A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTLNE4
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU STARTED THE LINE TO A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTLNE5
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU STARTED THE LINE TO A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTLNE6
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU STARTED THE LINE TC A REMOTE DEVICE CORRECTLY.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTI
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY THIS STARTS ALL THE INITIATORS.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTI1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTI2
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTI3
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTI4
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
```

```
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTI5
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTI6
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
CA $STARTI7
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY A PARTICULAR INITIATOR WILL BE STARTED.
TY VERY GOOD. YOU SEEM TO UNDERSTAND THE $S COMMAND.
WA(L) S&
TY YOU FORGOT THE '$' SIGN.
WA $SRM1PR
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM1PR1
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $SRM1RD
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM1RD1
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $SRM1.PR
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM1.RD
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM2PR
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM2PR1
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $SRM2RD
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM2RD1
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $SRM2.PR
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM2.RD
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM3PR
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM3PR1
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $SRM3RD
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM3RD1
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
```

```
WA $SRM3.PR
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM3.RD
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM4PR
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM4PR1
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $SRM4RD
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM4RD1
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $SRM4.PR
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM4.RD
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM5PR
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM5PR1
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $SRM5RD
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM5RD1
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $SRM5.PR
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM5.RD
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM6PR
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM6PR1
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $SRM6RD
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM6RD1
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $SRM6.PR
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM6.RD
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRMPR
TY WHICH REMOTE ARE YOU REFERRING TO?
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRMPR1
TY WHICH REMOTE ARE YOU REFERRING TO?
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $SRMRD
TY WHICH REMOTE ARE YOU REFERRING TO?
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
```

```
WA $SRMRD1
TY WHICH REMOTE ARE YOU REFERRING TO?
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $SRM.PR
TY WHICH REMOTE ARE YOU REFERRING TO?
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRM.RD
TY WHICH REMOTE ARE YOU REFERRING TO?
TY YOU FORGOT THE DEVICE NUMBER.
WA $SPRT
TY YOU FORGOT THE DEVICE NUMBER.
WA $SRDR
TY YOU FORGOT THE DEVICE NUMBER.
WA $SLNE
TY WHICH REMOTE LINE ARE YOU STARTING?
WA $STARTRM1PR
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM1PR1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $STARTRM1RD
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM1RD1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $STARTRM1.PR
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM1.RD
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM2PR
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM2PR1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $STARTRM2RD
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM2RD1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
```

```
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $STARTRM2.PR
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM2.RD
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM3PR
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM3PR1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $STARTRM3RD
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM3RD1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $STARTRM3.PR
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM3.RD
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM4PR
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM4PR1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $STARTRM4RD
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM4RD1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $STARTRM4.PR
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
    OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
```

```
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM4.RD
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM5PR
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM5PR1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $STARTRM5RD
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM5RD1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $STARTRM5.PR
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM5.RD
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM6PR
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM6PR1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $STARTRM6RD
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM6RD1
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE PERIOD IN THE REMOTE DEVICE NAME.
WA $STARTRM6.PR
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE DEVICE NUMBER.
WA $STARTRM6.RD
TY YOU CAN SAVE YOURSELF A LOT OF TROUBLE BY USING $S INSTEAD
   OF $START.  PLEASE USE THE ABBREVIATED FORM FROM NOW ON.
TY YOU FORGOT THE DEVICE NUMBER.
```

APPENDIX C

JOB CONTROL LANGUAGE NEEDED FOR OPERATION ON THE IBM 360/65 AT UTCC

Executing the SLATE program requires of minimum of four job control language (JCL) cards. These cards are:

1. The JOB card specifying the jobname and system accounting information.

2. The PROCLIB card which designates the dataset name to be read for further control information.

3. The EXEC card which names the program to execute.

4. The INCAI card which specifies the dataset name of the input.

A tape request card (//*TAPE) may be needed if the output is to be placed on tape.

There are several options available to the user when executing the SLATE program. For execution in a test mode, the output dataset may be allocated to temporary disk space. If so, the output will be deleted when the job terminates. If the data is to be maintained, the program will automatically request a nonlabelled tape for the output. The options are specified on the EXEC card if default options are to be overridden. A complete list of option keywords, meanings, and default values is given below:

Keywords	Meaning	Default
CAIREG	program region size	96K
PRIME	primary space allocation	10 blocks
SECDRY	secondary space allocation	10 blocks
DISP	output dataset disposition	KEEP
UNIT	output device	TAPE9
FILE	number of file on output tape	1

The default is a request for a tape with output beginning on the first file of that tape.

Here is an example of a typical JCL sequence to run the SLATE program using card input and storing the output on tape:

```
//jobname JOB (project code,account code),programmer name
//*TAPE         CAITEXT,W
//PROCLIB DD DSN=J34671.P45438.CAI360,DISP=SHR
//STEP1 EXEC CAIGEN
//INCAI DD *
     card input
```

For temporary storage of output data, using a catalogued input dataset called INDATA, these JCL cards would be used (assume 128K will be needed):

```
//jobname JOB (project code,account code,R-128K),programmer name
//PROCLIB DD DSN=J34671.P45438.CAI360,DISP=SHR
//STEP1 EXEC CAIGEN,CAIREG=128K,UNIT=WORK,DISP=DELETE
//INCAI DD DSN=INDATA, DISP=SHR
```

If more than one course segment is to be inserted into COURSEWRITER III, the SLATE program must be run for each segment. This can be accomplished in one job stream, though. Assume that two course segments are to be processed using card input and tape output. The following JCL would be used:

```
//jobname JOB (project code,account code),programmer name
//*TAPE         CAITEXT,W
//PROCLIB DD DSN=J34671.P45438.CAI360,DISP=SHR
//STEP1 EXEC CAIGEN
//INCAI DD *
     card input
//STEP2 EXEC CAIGEN,FILE=2
//INCAI DD *
     card input
```

The FILE=2 parameter specifies that the output is to be written after the output of step one. If more course segments were added the FILE parameter would be incremented by 1 for each segment.

Whenever SLATE is executed and output is to be saved on tape, the user must submit a UTCC Rental Tape Request Card (pink card) so that the tape will not be returned to the free tape area. In addition the user should contact the COURSEWRITER consultant in UTCC User Services for assistance with the AUTO INSERT function.

VITA

Alan Vincent Stone was born in Honolulu, Hawaii on August 22, 1952. He attended elementary and secondary schools in Cincinnati, Ohio and was graduated from Oak Hills High School in June, 1970. The following September he entered the University of Tennessee, Knoxville, and in June, 1974 he received a Bachelor of Science degree in Industrial Management-Management Science. In the fall of 1974, he accepted a teaching assistantship at the University of Tennessee, Knoxville and began study toward a Master's degree in Computer Science. During this time he also worked as an operator of the University's IBM 360/65 computer at the University of Tennessee Computing Center. The Masters of Science Degree is to be awarded in August, 1976.

The author is currently seeking employment status with a large corporation as a programmer/analyst and is to be married to Miss Kandy Callicott in January, 1977.