THE DESIGN, IMPLEMENTATION AND USE OF A COMPUTER-ASSISTED INSTRUCTION SYSTEM.

A Thesis

submitted to the

UNIVERSITY OF NEWCASTLE UPON TYNE

for the Degree

of

DOCTOR OF PHILOSOPHY

M. W. DOWSEY

SEPTEMBER, 1970.

PAGE NUMBERS CLOSE TO THE EDGE OF THE PAGE
SOME ARE CUT OFF

ABSTRACT.

Computer-assisted instruction has developed over the last decade from simple teaching machine principles to a wide variety of instructional strategies. This development has changed the characteristics of the instructional system and the skills of the people involved.

Design considerations for computer-assisted instruction systems are discussed, current operational systems are described and the need for another system is explained. In particular, attributes of author languages are discussed with reference to those in use at the present time and the author language which was embedded in the instructional system is described. How the system was implemented under a general-purpose time-sharing system is described, together with possible modifications and additions.

An investigation was carried out using the instructional system with two aims in view. Firstly, an attempt was made to validate the system and provide an appraisal of the facilities and services. Secondly, it was desired to study the teaching of programming languages by various methods. The methods employed allowed the comparisons of conventional teaching to teaching using computer-assisted instruction and demonstration classes to on-line examples classes to be made. The students' performance data is discussed and suggestions are made for future investigations of this kind.

In contrast to the use of author languages, which, being programming languages, some authors find difficult to learn, an easy author entry system was designed. This allows entry of course material in English on planning forms and provides self-documentation for the author.

Further details of the instructional system, together with examples of source code in the author language, student dialogue and student performance information, are contained in the appendices.

ACKNOWLEDGEMENTS.

I am indebted to the Science Research Council for awarding me a Research Studentship to enable me to study at the University of Newcastle upon Tyne from October 1967 to September 1970 and to Professor E. S. Page for admitting me to the Computing Laboratory to carry out research for a Ph.D.

I wish to express my sincere thanks to Mr. L. B. Wilson for his invaluable advice and guidance throughout the research project and in preparation of the text.

Also, I wish to thank all those persons who helped to make the investigation possible. They include Professor E. S. Page, who gave permission for the course to take place in the form it did; Miss E. D. Barraclough, who made the terminals and machine time available; Messrs. J. S. Clowes, M. J. Elphick and L. B. Wilson, who helped in preparation of the course meterial; Mr. J. F. Dunn, who helped with the NUTS/PIL interface; Mr. L. Waller, who made available the performance statistics; and Messrs. T. Anderson, D. R. Appleton, P. Henderson and J. L. Lloyd, who helped check out the instructional programs.

CONTENTS.

CHAPTER 1 Introduction

1.1 Computer-assisted instruction

The effects of the population explosion together with the tremendous growth of science and technology in recent years have presented increasing challenges to educators in schools, universities and industry. In addition, sociological changes including higher educational requirements for all students and increased emphasis on personality development have added to the responsibilities of teachers. To keep pace with these developments, new ways have been sought to help teachers absorb these new responsibilities while continuing to maintain high standards of academic achievement. These efforts have inspired the development and use of many new instructional techniques. Among these are audio/visual aids, instructional films, educational television, programmed instruction (PI) and, most recently, computer-assisted instruction (CAI).

The basic principle of CAI is that each student can learn a concept or subject according to his own particular requirements. This not only means that each student can learn as quickly as possible, but also that the amount and content of material presented can be tailored to his individual needs. Using this system, students who are able to grasp and retain subject matter with a minimum amount of explanation can advance to new material in accordance with their ability. Slower students can be directed to alternative presentations and exercises that allow them to learn at a slower pace. The teacher can obtain performance records that indicate which questions were answered correctly or incorrectly, whether any unanticipated responses were returned, actual response times, and other information to aid the evaluation of the individual student's performance. This makes it possible for a teacher to pinpoint areas where a student is experiencing difficulty with course material. He can then take remedial action by giving special attention to the student involved or by revising the presentation if analysis shows that performance of many students is low.

1.2 The emergence of computer-assisted instruction

Teaching machines were patented as early as 1809 (Mellan, 1936) but the current interest in teaching machines is generally traced back to Pressey, a psychologist at Ohio State University, who in the 1920's built several ingenious machines that automatically tested students by presenting a series of multiple-choice questions, printed on a roll of paper, which could be answered by pushing one of four keys. If the student was right, the machine presented the next question. If he was wrong, the machine recorded the fact and required the student to try again. The machine was small, quite simple and admirably served the testing purpose Pressey had in mind.

Pressey received little encouragement from the educational world for his ideas, and he indicated in 1932 that he was regretfully dropping further work while hoping that he had done enough to stimulate other research workers.

Other workers were apparently not stimulated by Pressey's work and it was not carried on. It was not until twenty years later that the idea of teaching by machine was effectively put forward, this time by Skinner of Harvard University. Skinner (1954) proposed a method of teaching based largely on his many years of research using pigeons as subjects. In his approach, a key aspect of teaching was to reinforce correct responses as quickly as possible, by rewards of food in the case of pigeons. Skinner believed that reinforcement could be effective in human learning.

Accordingly, he developed a method of teaching in which students are required to answer a sequence of questions, each only a little more advanced than the previous question. The student finds out immediately whether or not his answer is correct, and since it almost always is because the steps are small, his response is presumably reinforced.

A few years later, a different approach to machine teaching was put

forward by Crowder (1960). This was "intrinsic programming", in which larger steps are taken in the programmed material. If the student assimilates the new material, he moves ahead. If not, he is "branched" to a remedial presentation.

The intrinsic program has two practical limitations which may be important. Firstly, the student cannot formulate his own answer to a question, but must choose one of the answers given. Secondly, the number of levels of branching feasible with a simple machine or with a more common "programmed textbook" is not very large. The number of individual paths increases as the product of the number of possible incorrect responses at each level of branching, and this proliferation of possible paths in branching programs presents a formidable obstacle to intrinsic programming using books or simple machines. In addition, branching can be based only on the student's last response.

Certain criteria developed for programmed instruction are now generally accepted. They are as follows (Silvern and Silvern, 1966b):

(i) instruction is provided without presence or intervention by a human instructor.

(ii) the student learns at his own rate, as opposed to films, television, conventional group instruction, etc.

(iii) instruction is presented in small incremental steps requiring frequent responses by the student; step size is a function of subject matter and characteristics of the student population.

(iv) there is a two-way communication between student and instructional program.

(v) the student receives immediate feedback informing him of his progress.

(vi) reinforcement is used to strengthen learning.

(vii) the sequence of lessons is carefully controlled and consistent.

(viii) the instructional program shapes and controls the student behaviour.

A substantial amount of research has been done on simple linear programs but much less on intrinsic programs. A review of research results up to early 1963 (Schramm, 1964) indicates that experimental evidence about the effectiveness of programmed instruction techniques studied is not very clear-cut.

Schramm found 36 reports of research in which programs were compared directly with conventional classroom instruction. Of these 36 studies, 18 showed no significant difference, 17 showed some superiority for the programmed course, and one showed superiority for conventional teaching. However, Schramm ponders over the problem of what kind of teacher is being compared with what kind of program and discusses the novelty effects of such features, but admits that it is almost impossible to make allowances for novelty.

This and other studies of programmed instruction show two things with reasonable certainty. Firstly, the hopes of early proponents of programming that the process would teach more effectively than human teachers have not been demonstrated convincingly. Secondly, Pressey's original idea, that programmed teaching could free the human teacher from a good deal of the drudgery of presenting straightforward factual material, and testing students on it, appears to be quite valid.

In the late 1950's, a number of people recognised that computers might be ideally suited to programmed instruction, since they could overcome the severe limitations of the simple machines or programmed textbooks that were being tried. Computers can accept and evaluate responses constructed by the student, can provide almost unlimited branching capabilities, and can branch based on a variety of criteria. They can also control a wide variety of terminal equipment and in other ways provide far greater flexibility than is possible with simple teaching machines.

Perhaps the first CAI experiment was carried out at the IBM Watson

Research Laboratory (Rath et al., 1960) in 1958. Other early CAI studies were performed by the University of Illinois, System Development Corporation and Bolt, Beranek and Newman, Inc.

The potential of computers for instruction was summarised by Uttal (1962) who suggested that the computer teaching machine concept is quite different from the linearly programmed teaching machines. The flexibility and range of capabilities offered to the teacher by the powerful decision logic and large memory of computers transcend mere quantitative differences and suggest that there is a true qualitative difference between the two.

Some of the qualitative differences envisioned at that time were:

(i) the computer can continuously compare the student's performance with criteria established by the teacher, and present new material, or direct the student to the teacher or to the library, depending on his performance. "Performance" can be judged on the basis of a rather large number of individual responses by the student.

(ii) the ability to do what amounts to continuous testing can be used to prevent a student's moving ahead before he has mastered the material at hand, as well as preventing his being held back by the constraints imposed by classroom teaching.

(iii) the computer can keep quite elaborate records of the details of each student's progress, and the teacher can query the machine about any student at any time. Ideally, the teacher can be far better informed about his students' progress than is possible in an ordinary classroom situation or with simple teaching machines.

(iv) since course material is stored in a computer's memory, it is easy to change. Thus computer courses can be improved quite easily with experience, while it is relatively difficult to change teaching machine programs or programmed

textbooks once they have been printed.

The early experiments in CAI used rather simple programs and presented students with what was probably the most pedantic teacher of all time. Such trivial differences as double-spacing between words or missing commas between the student's response and the specified answer caused the machine to mark the student wrong.

These experiments showed that it was essential to develop computer programs that could ignore minor errors or variations in wording in a student's response and somehow determine its meaning. It was also recognised at this time that flexible and convenient student terminals and a programming language that would make possible the writing of courses by people who knew little about computer programming were essential for long-range development of CAI.

1.3 Methods of using the computer

There are two modes in which computer based systems are currently being used. Firstly, there is the direct mode in which the teaching actually takes place at a terminal. There are a number of methods which fall into this mode. Secondly, there is the indirect mode which makes less demand on computer facilities and is based upon broader applications of programmed learning principles, capitalising on existing teaching and training instruments rather than attempting to replace them. CAI does not generally embrace this latter mode, but for completeness, brief descriptions of the two methods from the mode appear as the latter two in the following descriptions of methods. The first four described are the most widely used and probably the most important.

1.3.1 Drill and practice

This is usually associated with presentation of basic skills and is probably the most easily programmed. Drill is a technique to supplement the student's knowledge by supplying him with common facts and skills. Practice allows him to use these facts and skills until a certain degree of proficiency is achieved. If it is a linear program, all students negotiate the same pattern of questions. The only individualisation is in rate of completion. However, if adaptive, the program may present graded sets of exercises, and by comparison and response latency, it presents the student with selected feed-back and adapts the sequencing to fit the individual. The computer also records a complete performance history.

1.3.2 Author-controlled tutorial

Here, information is first presented to the student and then he is required to respond to a question based on the facts just received. The system judges the answers by comparison with expected responses. In an intrinsic program, usually made up of a sequence of multiple choice items, the selection of each presentation is based on the student's response to the

previous stimulus. Thus, by branching, each student may take a different path through the subject matter. The program is adaptive in pace and in the amount of instruction each student receives. In an adaptive tutorial program, each presentation is based on an extensive history of student responses to the program. In either case, the author of the material maintains the initiative throughout and plans an optimum path through the program which he expects the better students to follow.

1.3.3 Socratic tutorial

The Socratic method is characterised by the program permitting dialogues between the student and the computer. It goes beyond tutorial logics by allowing the student to assert an answer or solution at any point in the interaction, or to ask for data. The author retains considerable control over the student's behaviour as he usually makes the computer messages dependent not only on the student's last response but also on the history of the conversation. However, he must be prepared to return a meaningful reply to almost any reasonable request the student may make. The author may sequence the availability of information during the inquiry stage and may guide the student towards an acceptable solution in the decision-making stage.

1.3.4 Learner control

This relatively new logic originated by Grubb (1968) places the initiative with the student, who accesses only the subject matter which interests him and hence learns by discovery. This allows the students to approach the subject in different ways, from those who only want to browse through the course to those who seek specific information. Originally, an outline map is presented to the student. This contains the structure of the course material indicating the various concepts and topics involved. On choosing a particular concept, the student is confronted with a further, more detailed map of this area. At the lowest level, instructional material is presented to the student but at any point in time the student may

branch forward to skip over a question, branch backwards to repeat a presentation, jump to a glossary for definitions, or jump to the current level of subject outline if he is in too deep. Comparison studies of this technique have been made (Grubb, 1969).

1.3.5 Simulation and gaming

The logic behind simulation is to duplicate in the learning situation the format and sequence of stimulus events in the real world. It can provide experience for the student under conditions of greater safety, greater economy or with great savings of time. Gaming differs from simulation in that there need be no real situation. Usually, there is an element of competition, particularly involving groups of students. The program may allow the student to work in a self-instructional mode, and provide basic tuition, automatically setting up problems for him to solve, giving specific assistance where necessary and assessing his performance.

1.3.6 Paired students

In an effort to combat the fact that during a CAI session the student learns in isolation yet the classroom is potentially wealthy in dynamic social interactions, Grubb (1965) carried out a study on the effects of paired student interaction. Little work, however, has been done in this direction.

1.3.7 Test and assessment

This method is an indirect interaction and hence is not normally regarded as being CAI. The computer is used from time to time to administer criterion tests to measure whether educational objectives are being met. During the test, a correct response is met with immediate reinforcement whereas an incorrect response causes further questioning in an effort to discover exactly what difficulties are occurring. The computer assembles information to assist the teacher in preparing and scheduling future classwork.

1.3.8 Computer-managed instruction

This is management of the learning process, not presentation of the instructional material itself. Usually, there is a well structured series of work assignments or activities which may call on numerous different learning resources and which may include many parallel options. After each activity, the system tests the student on-line and hence determines his next assignment, either appropriate remedial work or further study. The teacher is provided with the analysis of the test data and the sequence of activities to indicate the progress of each individual student. A description of a project of this type is given by Gilligan (1969).

1.4 Personnel required for computer-assisted instruction.

In comparison to conventional teaching where each class is taught and supervised by one teacher at any point in time, numerous personnel, each gifted in some particular skill, are required for a CAI project of any size. Quite naturally, the number of students that such a project would support is increased in proportion.

Just exactly how many different members of staff are required and what their speciality should be has always been a matter for conjecture.

Silvern and Silvern (1966a) suggest that there should be:

(i) the teacher, who becomes the manager of the education process and evaluates and counsels students with the help of reports from the computer. He provides special and remedial instruction and is thus elevated from the present position of communicator to that of directing communications.

(ii) the instructional programmer, or author, who, in fact, may be a team of persons or one senior individual with a staff of specialised assistants. He initially performs a job and task analysis, then proceeds to establish behavioural objectives, devise criterion tests to measure these, develop the course outline and finally write the steps in the lesson plan.

(iii) the computer systems programmer, who may be expected to write the CAI compiler, integrate the processor into the operating system or add capabilities beyond the normal CAI language.

(iv) the computer operator (or proctor as he is sometimes called), who looks after the running of the machine and its peripherals, and helps students use the equipment.

Zinn (1968) proposes that the job of the instructional programmer, as defined above, should actually be divided into three separate positions:

(i) instructors, who select programmed strategies into which they need only enter the teaching material and some answer-

processing rules. They will be managers of self-instruction yet not expected to have special knowledge of computer programming.

(ii) authors of instructional strategies, simulations or academic games, who, using a special-purpose language, provide a basic strategy and organisation of content which an instructor might later modify in superficial ways.

(iii) instructional researchers, who should be able to invent special strategies to switch from one instructional mode to another, accumulating and comparing data on performance of different students.

In the light of the experience of a large scale physics project, Hansen (1970) describes the various roles which evolved:

(i) content scholars, who prepared a detailed conceptual outline of the course.

(ii) behavioural scientists, who provided criteria for the behavioural consequences of the instruction and analysed the issues dealing with the topics of entry behaviours, task analysis, behavioural objectives and instructional strategies.

(iii) physics writers, who, since the talents of the two preceding groups were in short supply, performed the detailed writing of the instructional materials.

(iv) CAI coders, who entered the instructional material into the CAI system, using the author language.

(v) media specialists, who helped prepare the concept films and audio tapes.

(vi) computer operators, who supervised the running of the machine.

(vii) computer systems programmers, who developed a data analysis and management system for the project.

(viii) a data analysis programmer, who amended statistical programs to provide performance reports.

(ix) CAI proctors, who assisted students in preparing the multi-media devices for utilisation.

(x) graduate students, who acted as demonstrators for the course and raised queries about the overall systems approach, including strategy and media selection.

The need for such an array of staff naturally raises the question of cost of CAI. A thorough discussion of this aspect, and comparison with the conventional method of teaching is given by Kopstein and Seidel (1967).

1.5 Layout of this thesis

The remainder of this thesis is divided into four sections, three of which are self-contained. Chapter 2 gives a detailed description of the teaching system that was designed and implemented in Newcastle, complete with a description of other systems. Chapter 3 describes an investigation which was designed to provide a validation and evaluation of the system and determine how best it can be used in the area of teaching programming languages. Chapter 4 turns away from the idea of generating instructional material via an author language and suggests a method by which the vast team mentioned in 1.4 can be replaced effectively. The final section provides a summary together with conclusions of the whole study.

In the appendices are given the user's guide to the teaching system, a sample of the author language coding used in the investigation, a sample student dialogue with the course and the performance record resulting from this dialogue.

CHAPTER 2 The Newcastle University Teaching System (NUTS)

2.1 Computer-assisted instruction systems

2.1.1 Introduction

The first attempts at producing CAI programs started over a decade ago and used computer systems and programming languages currently in use at that time. Only one user at a time could use the machine, whether he was the author developing the course or one of the students receiving the instructional material. The fact that an existing programming language was used to code the programs probably meant that the response processing was the least sophisticated ever attempted. When the student responded to a question from the computer, he was judged wrong by the machine if he made such trivial mistakes as double spacing between words or leaving out a comma.

Perhaps the first experiment with computers for instruction was carried out by Rath et al.(1960) in 1958 at the IBM Research Centre using an IBM 650 computer to teach binary arithmetic. Other early CAI experiments were carried out at the University of Illinois, System Development Corporation and Bolt, Beranek and Newman, Inc.

Since that time, there has been a general trend towards a specialised CAI system. Instructional programs are assembled into a format that can easily be executed by an interpreter in real time during instructional sessions. In addition, background programs permit users to schedule card assemblies, list courses and student records, load functions, macros, dictionaries, graphics, etc.

However, in more recent years, CAI subsystems running under general-purpose time-sharing systems are being developed.

2.1.2 Early operational systems

During the period 1962-65, CAI systems grew, more or less independently, in several laboratories. These pioneering examples of CAI shared some common features, such as time-sharing. Each also had unique features. In this section, several well-established systems of that period are listed and described.

In that time, each of the systems was restricted to the laboratory or campus on which it was generated. However, in 1965, remote terminals for several of these systems were installed at other laboratories and universities, sometimes many miles distant. Thus, some systems became no longer identifiable solely with a particular institution, except historically or administratively.

2.1.2.1 IBM Yorktown. An extensive system was based on the IBM Watson Research Laboratory at Yorktown Heights, New York. This system originally used an IBM 650 RAMAC with 20 terminals and a disk file for 6 million bytes and 0.8 seconds maximum access time (Grubb and Selfridge, 1963). In 1964, an IBM 1440 replaced the 650.

Students at Pennsylvania State and Florida State Universities took courses transmitted from the Yorktown complex (Wodtke et al., 1965). The course was presented at a modified electric typewriter and used a random-access slide projector and a tape-recorder.

2.1.2.2 IBM Poughkeepsie. A system similar to the Yorktown system was based on an IBM 1440 at Poughkeepsie. Through 12 IBM 1050 terminals located at IBM offices at Poughkeepsie, Los Angeles, San Francisco, and Washington D.C., IBM customer engineers received training while on call at their offices.

2.1.2.3 CLASS. The Computer-based Laboratory for Automated School System (CLASS), developed by the System Development Corporation, provided instruction for up to 20 students under the control of a computer. It was designed to study branching effects and permitted the investigation

of systems problems as well as individual learning processes in a controlled environment (Ruans, 1963). In addition to a practical course on statistical inference, CLASS was used for computer-based student counselling and field evaluation of an elementary Spanish course.

2.1.2.4 System 437L. System 437L was a command-control system operated by the U.S. Air Force. It included an automated instructional subsystem which taught Air Force console operators the query language to be used in communicating with the main system (Clapp et al., 1964). Because of obsolescence of the basic system, the training subsystem was not implemented.

2.1.2.5 SAKI. The best known of the machines developed in the U.K. by Pask (1959) was Solartron Automatic Keyboard Instructor (SAKI), which instructed the student in keyboard operation. Errors and response times were calculated by SAKI, compared with the standard performance, and fed back to the student. It was not a time-shared system.

2.1.2.6 COBIS. The Computer-Based Instructional System (COBIS), located at the Electronic System Division, Hanscon Air Force Base, Bedford, Mass., was based on a DEC PDP-1 (Baker, 1965). It had three principal features. Firstly, a light pencil was used as the medium of communication between the student and the computer; secondly, the student indicated his degree of certainty for each alternative in the multiple choice array by adjusting bars of light next to each answer on the cathode ray tube; and, finally, the computer considered both the student's answers and his degree of certainty when branching to remedial sequences or further steps. A special scoring system was developed accordingly.

2.1.2.7 The Socratic System. In the Socratic System, built by Bolt, Beranek and Newman, Inc., in April 1963 around a DEC PDP-1, the teacher and student carry on a dialogue in depth (Swets and Feurzeig, 1965).

The computer states the problem, sets up conditions, asks questions, provides requested data, and answers questions, while observing the student's course of action in a task. The system has been applied to instruction in medical diagnosis and business decision making.

2.1.2.8 SOCRATES. SOCRATES was a time-shared system developed at the Training Research Laboratory (TRL) of the University of Illinois (Stolurow, 1965b). It was adaptive in three ways. Firstly, it learnt about the student as it taught him. Secondly, it might make decisions about the effectiveness of the rules used to teach the student; and, finally, it might make decisions about the criteria which were used for evaluating performance.

The SOCRATES I student interface was put on-line in May 1964. Five terminals were available during the summer of that year. It used an IBM RAMAC external disk memory of 2 million-byte capacity to supplement the internal storage of an IBM 1620 computer.

In SOCRATES II, an IBM 1311 disk replaced the RAMAC disk and different terminals were used. The new terminals displayed not only 35 mm. frames, but also a set of six fixed messages, each of which would be selectively called by the program. In addition, these student terminals provided random access to any of 1500 frames of film and contained a keyboard consisting of 15 keys with interchangeable characters.

SOCRATES software permitted the storage and use of historical information by student rather than by terminal. The student file permitted full use of the idiographic model. SOCRATES provided records of every response of each student, in terms of both the time it took and the character of the response.

The project was discontinued late in 1966.

2.1.2.9 PLATO. At the Computer-based Education Research Laboratory (CERL) of the University of Illinois, a single-student version of the PLATO (Programmed Logic for Automatic Teaching Operations) system was used to present a variety of subject matter ranging from mathematics to French grammar. PLATO II, the first multiple-student teaching device from the laboratory, used the ILLIAC computer with a high-speed memory of only 1024 words, which limited the system to two terminals. The "electronic-blackboard"

television display technique has both a cathode ray tube and a slide display superimposed. Such function keys as "help" and "aha!" are also available to the student.

In 1964, transition was made to PLATO III, based on the CDC 1604 computer. PLATO III had a theoretical limit of 1000 terminals, but only 20 were implemented. The "help" sequences were increased from one to eight. Analysis of the student performance records on-line became available and intercommunication between terminals, added in 1965, made experiments in gaming, simulation and group interaction possible. A description of the PLATO system is given by Bitzer and Easley (1965).

2.1.3 Design considerations

The design of a CAI system, quite naturally, depends to a large extent upon the type of system that is required. For the most part, CAI systems are implemented upon general-purpose computers but they are a separate entity, apart from any other subsystem available on the machine. However, there exist great advantages in constructing CAI subsystems which are able to transfer control back to the operating system to obtain any other subsystem such as a compiler, simulator, etc., before returning to the instructional subsystem. Finally, there is a class of CAI systems which is highly specialised. For instance, special consideration may be made as to the method of course assembly, type of instructional strategy, or method and reason for use by the student.

2.1.3.1 General stand-alone systems. The minimum requirements and desirable characteristics of an instructional system aided by a general-purpose computer are given by Zinn (1965). He considers the six lines of communication between student, learning materials and author and to each of these six lines attributes one requirement:

(i) development of an author language for material presentation and strategy definition;

(ii) analysis of material, strategy and student performance;

(iii) display of material;

(iv) processing of student responses;

(v) furnishing of unanticipated student requests; and

(vi) provision of individualised instruction.

Tonge (1968) suggests that any new system should also allow student-course interactions of the type already familiar in other systems, as above, but suggests further that the system should

(i) permit the entry of course programs either on-line or using batch input devices and should allow the author to correct and immediately test material on-line, even during student usage of other parts of the same course;

(ii) provide to the course author capabilities for calculation, test analysis, data base access, and abbreviated reference to commonly used sequences of material (a macro facility);

(iii) allow "easy" modification of language syntax and semantics so as to encourage course authors to consider and suggest language improvements;

(iv) facilitate experimentation by authors with more sophisticated algorithms for response processing;

(v) provide a computational facility to students, in the context of the course program; and

(vi) be implemented so as to maximise the efficiency of the highly repetitive student interactions rather than the less frequent author debugging sessions.

Adams (1967) reinforces most of these considerations but, in addition, points out that

(i) the system response should be "very fast", about a tenth of the time it took the student to frame the message;

(ii) as well as both on-line and off-line entry of programs, there should be a very fast compile service where small alterations are made to a program;

(iii) there should be a capability to mark and specify the contents of transaction records which may include time data, contents of storage and the text of messages;

(iv) the system should offer a recovery from malfunction; and

(v) the system should be open-ended, i.e. capable of executing special routines written in a lower-level language and adding new routines to the source language at will.

As for implementation, Silvern and Silvern (1966a) suggest that the basic CAI system be machine independent and written in a widely-used language such as FORTRAN, and, since the system would handle many students taking the same or different courses simultaneously in a time-sharing mode, it should use re-entrant coding. They also put forward the ideas that the sign-on procedures should be simple, yet give adequate security, and that a student should have no difficulty restarting a course after a previous session. Perhaps their most sensible suggestion, but probably the most difficult to implement, is that the system should respond with "WAIT" if there is a time lag of over three seconds when processing a response.

2.1.3.2 CAI systems as subsystems. Instructional systems may incorporate other programming facilities which can be used by both author and student. Already, a basic version of COURSEWRITER has been "married" with FORTRAN in a conversational, tutorial system called TUTOR at CERL, University of Illinois. The curriculum expert can set the problem for the student and provide some discussion in the tutorial mode. The student shifts to the edit mode to construct his solution and then calls the FORTRAN compiler or other system software to complete the job.

PLANIT, at System Development Corporation (Feingold, 1967), permits the author to specify statistics problems for which the computer generates data and determines the correct answer. The student can use the computer as a desk calculator, call on available subroutines and write simple

programs of his own.

In addition to simple computational aids, some lesson designers may want to provide an algebraic language, a text processing language, a model-building or simulation language, perhaps a specific system or model written for student use, or information organisation and retrieval capability.

A broadly conceived instructional system probably should begin with a general-purpose system and add the facility for moving from the terminal mode into other user subsystems, returning when an exercise is completed. Some authors need to maintain contact with the student through some means of monitoring his work on a problem. It may be necessary to bring him back to the tutorial mode because of elapsed time, number of problem attempts, or even anticipated error which requires special attention.

Engvold and Hughes (1968b) have designed such a subsystem. They point out that adding a number of functions to make a teaching system more flexible would require extensive additional programming and the number of such functions that could be added might be limited by the size of the computer memory for which the teaching system was designed. In order to improve man-computer communication, they made the computer controllable from a display unit. All the resources of the operating system can be summoned by having the user point a light pen at the display. Thus, these full resources (language processors, compilers, models and other library and user programs) can be called upon from the display in order to enrich the teaching process. In addition, the user can be instructed or guided in employing this software by exercising the system's tutorial function.

2.1.3.3 Special purpose CAI systems. We have mentioned that the design considerations of a CAI system may depend on some special feature of that system; for instance, the instructional strategy, the method of generation of material or the reasons for use of the system.

Stolurow (1965a) defines systems analysis as the construction of models and procedures to optimise some function of the variables involved in the model. He used an operational model of instruction to create SOCRATES (1965b). The model is ideomorphic in that it considers the learner's characteristics, and is itself adaptive during instruction. This involves three steps in a cycle. Firstly, the pretutorial step selects the optimum teaching program for each student; secondly, a specific set of instructions, task parameters, definition of tasks and rules is implemented; and, finally, the tutorial design is changed or a better set of rules is adopted to reach the desired performance.

A similar approach has been made at the Human Resources Research Office (Hum RRO) with project IMPACT (Kopstein, 1969). The philosophy there is that, if an instructional system is to be better than that of a good human instructional system, more predicative indices must be used. Starting from a simple (minded) instructional decision model empirical data is collected and fed back to modify the model. The current version is tested diagnostically to find indications to what is right and what is wrong with it and the model is then adjusted in terms of the data obtained. It considers such factors as the student's abilities and educational background, his fluctuating motivations, his cumulative patterns of progress toward mastery and his pattern of errors in recent criterion tests, and relates them in a logical and unvarying way to instructional options, such as the available subject matter information, the available media and the available methods or forms of presentation.

Several systems have been suggested, designed and implemented which emphasise the ease of generation of instructional material.

Meadow et al. (1968) produced a course generator, CG-1, an interactive program, which produces a course program as the result of a conversation between the computer and the course author carried on in natural language. The generated programs are in a language called PL/I Interactive Dialect

(PL/I ID) which is a dialect of PL/I and which can be compiled and run on most IBM System/360 computers.

The system due to Kerr et al. (1969) is quite different. In it, a course consists of units called frames. Frame elements are composed, punched into cards and then stored as records on a direct access storage device. The authors believe that run-time generation of these frames, which are actually stored as character strings, is a more suitable approach than material prepared in a special or general-purpose language and compiled for instructional use. Courses can be changed and expanded without changing the processing programs, and no computer program is required to include new courses in the system.

Whereas both previous systems have allowed for a wide range of question-types during the generated course, Uhr (1969) proposes a system in which the author must specify the general type of question he would like to ask. The program generates a sequence of questions in some particular problem domain during, and as a function of, its interactions with the particular student it is teaching. Rather than pre-programming the text into the computer, the teacher only codes, in a standard format, the type of question he wishes to ask. The program then generates a long sequence of particular questions, questions that become progressively more difficult as the student succeeds in answering the simpler ones, and branches back to simpler material when the student fails.

For the most part, CAI systems have been used directly in the teaching environment. However, Winkler (1968) suggests that the traditional forms of CAI are in the process of developing into a public "utility" of organised knowledge. He postulates a nation-wide network of computers which will be used by schools to supplement instruction and for placement and certification purposes. The system will also be used by libraries for browsing, by corporations for inservice training and personnel selection and by individuals as a device for learning what one needs to know in order to gain knowledge in particular fields. This follows from the different levels

of difficulty attributed to all the available CAI programs and the chain references from one to the next. One important technique suggested is that all computers would keep a listing on drum or disk of the CAI programs in the current inventory. The lack of a requested program in the inventory would cause the executive to dial the appropriate higher level to secure the program. This continues until the program is secured, from the national computer, if necessary. The CAI program, in being transmitted back to the requestor, is then stored in each level in anticipation of future use by this or some other requestor. Each "level" of request will, therefore, become a reservoir of CAI programs and the function of this reservoir is to reduce the number of requests sent to the next higher level by a factor of at least the ratio of the number of computers at the two levels.

2.1.4 Current systems

This section contains a summary of those CAI systems currently in use.

2.1.4.1 IBM System 1500. In the IBM 1500 Instructional System, different operations can be performed by students taking courses, teachers writing courses, and proctors supervising the overall operation of CAI. In addition, background jobs can be completed when the system is not being used for instructional purposes. The major functions of the operating system include:

(i) scheduling service requests so that each station is offered an opportunity to use the system facilities in turn;

(ii) directing proctor assistance to the students requiring it;

(iii) accumulating student records;

(iv) analysing and executing proctor instructions;

(v) providing information about system operation to the proctor when necessary; and

(vi) storing and maintaining all data needed by the programs under operating system control.

The significant characteristics of the operating system are that it is a re-entrant, natural-interrupt, terminal-oriented, time-sharing system. It is divided into five major sections: station IOCS, scheduler, command processor, service routines and applications. The system can be based on either an IBM 1130 or 1800 and with 32K core can support up to 32 stations. A system summary is given in an IBM (1967) manual.

2.1.4.2 RCA Instructional 70. Lesson material is organised as a series of concept blocks, each of which provides drill and review material to students at various levels of difficulty. The computer is programmed to present these concept blocks in a specific sequence but the teacher may change this sequence to parallel his own presentation. The computer also produces two reports for teachers: the daily status report and the concept block progress report. The daily status reports are produced after each day's student sessions; one report is produced for each class. The concept block progress reports provide detailed information about each student's progress within a concept block. The programming elements of the system include:

(i) the Instruction Systems Language - 1 (ISL-1);

(ii) the text editor (used to create the files of data constituting the curriculum data base);

(iii) the data translater (used to translate material from the text editor into the format expected by the 70 system procedure program);

(iv) the operating system; and

(v) the data management system.

The operating system controls the on-line instructional process, and is divided into four subsystems: the control monitor, interpreter, communications control system and disk control system. The data management system produces the off-line progress reports. A Spectra 70/45 processor unit is used with a 262144 - byte core storage and each line concentrator may service up to 48 terminals. General information is found in an RCA(1967)

report.

2.1.4.3 RCA Instructional 71. The four main functions of the operating system are:

(i) operation of the central processor, auxiliary storage units, operator's console, and communication interface;

(ii) file maintainance after instruction has ceased;

(iii) entrance of the curriculum materials into the system using ISL-1; and

(iv) translation of ISL-1 procedure programs into a machine-oriented format.

The system is controlled by an RCA 716 CPU, which has 65536 bytes of high-speed core and can handle as few as 14 and as many as 48 students, depending upon system configuration. A description is given in an RCA (1968) report.

2.1.4.4 Technomics 6700. This system has three major parts: the central computer, the teaching consoles and DIALOG, the lesson-unit compiler. DIALOG, the most important part, allows the teacher and student to converse with the system in natural English. The task of writing a program has been reduced to a relatively small set of choices or decisions. The CPU is a 16-bit machine with a core memory of 16000 words expandable to twice that size. In the basic system, 30 consoles may be simultaneously in use but with access to a large computer, the system may be expanded to 50 or 75 consoles. The console has a television-tube display on which video pictures, line drawings and written text may be presented (Hickey, 1968).

2.1.4.5 PLATO. The operating system for PLATO is based on a CDC 1604 and includes 20 teaching stations with video capability. The author may project slides on a television screen via an "electronic book" and superimpose writings or diagrams by means of an "electronic blackboard" function. Student response is by teletype keyboard with user-defined characters or special symbols appearing on the television screen at a

location predetermined by the author or programmer (Bitzer and Easley, 1965). A special system for response analysis (Easley, 1967) provides general facility for retrieval and review of records on a visual display. The author is able to review a trace of the student's progress through an instructional sequence, obtaining summary statistics at various levels of detail, or even replay at a student console a complete interaction. In this latter case he may also specify the relative speed that is required compared to the original. The software to specify the logical structure of an instructional sequence is written in an extended FORTRAN for the PLATO compiler, CATO (Compiler for Automatic Teaching Operation). Considerable flexibility is allowed the user familiar with the three levels of language: CATO, FORTRAN and assembly. In 1970, a large computer capable of controlling more than 4000 stations will be ordered to reach the goal of 4096 stations in 1974, by which time it is expected that the cost per student hour for terminal and CPU time will be less than 25 cents.

2.1.4.6 The Socratic System. For the "Socratic System" (Feurzeig, 1965), a language called MENTOR was developed for authors to use in constructing conversational, tutorial dialogues. The author of materials may, via computer storage and logic, take the role of advisor, monitor, interviewer, consultant, examiner or tutor. Typically, a situation is established in which a problem may be solved by the gradual acquisition of information. The student types enquiries of declarations selected from a list of acceptable terms, and the machine identifies these even when misspelling occurs and types an appropriate reply according to complex conditional statements provided by the author. The system uses teletypes attached to a modified PDP-1. MENTOR is written in LISP, and sections of instructional programs otherwise coded in MENTOR may be written in LISP. The same system is used with an on-line computational language, TELCOMP, for instruction in schools.

2.1.4.7 PLANIT. The initial design started in January 1966; by June, PLANIT (Feingold, 1967) was operational. It was written in JOVIAL (Perstein, 1966) and used an IBM AN/FSQ-32V computer via an interactive console under the SDC time-sharing system. The user (lesson designer or student) communicates with the system via a keyboard device linked by either telex or telephone to the computer. PLANIT comprises not only the author language but also a program developed for time-shared use. The system operates in four modes: lesson building, editing, execution and calculation. The first two modes permit the author to construct and edit lesson frames in various formats and store them in designated sequences for later presentation to the student in the execution mode. The calculation mode is particularly oriented to mathematical subject matter and can be used as a calculation aid for the author (when building the lesson) or the student (when performing the lesson). While the student has access only to execution and calculation modes, the author may use all four. PLANIT allows one lesson to call another, and any program (or subroutine) written in JOVIAL can be added to the lesson and executed at any time.

2.1.4.8 CAL (Irvine). The CAL (Computer-Assisted Learning) system (Tonge, 1968) at the University of California, Irvine, is implemented on an IBM 360 model 50 under an Interactive Application Supervisor (Summers et al., 1967) which furnishes the scheduling algorithm, terminal control and file-handling capabilities for all subsystems. The supervisor provides standard editing features and conventions for all subsystems, allowing backspace and type over, underlining and so forth. The components of the CAL system include table driven syntactic and semantic analysers, an assembler, a pseudo machine (interpreter) for processing assembled student programs, and interface routines for communicating with the student and course author. Two of the more interesting basic files are the error file and the response log.

The error file contains a list of error messages as they occur during the day, including the student context at the time of the error so that authors and system programmers may analyse and correct error situations. The response log contains the record of all student responses and appropriate student context information, as requested by course authors for later analysis.

2.1.4.9 The Leeds system. The Leeds system has been implemented on an Elliott 903C computer, which has 8K words of main store, a paper tape punch, a paper tape reader and an on-line teletype (Sleeman and Hartley, 1968). For each session, the processor, which controls the learning process, has to be read into the machine through the paper tape reader and at the beginning of each different lesson the appropriate teaching material has to be put into the remainder of the main store in a similar manner. One important feature of the system is that, during instruction, the student can have access to files which are stored in the computer provided that he types the proper request on the teletype (Hartley and Sleeman, 1968). Exactly which files are set up depend upon the nature of the problem, but they are always referenced by function and include such requests as FACTS, for information; MEANINGS, from which definitions, symbols or formulae can be obtained; EXAMPLE or TEST, to provide practice on the use of the meanings; CALCULATE, for numerical capability; and HELP. The Elliott 903C has now been replaced by a MODULAR-1.

2.1.4.10 ADEPT. The ADEPT (A Display Expedited Processing and Tutorial) System was developed experimentally for the IBM 360 model 40 and the 2250 display unit, model 1 (Engvold and Hughes, 1968a). It operates under OS 360 and the machine used has 256K storage, 3 disk storage devices and 4 magnetic tape units. The 2250 has an 8K buffer and a light-pen, alphameric keyboard and a program function keyboard. All secondary storage resides on disk. The system has three main parts:

(i) a language containing control and text codes which operate in one of either the author mode, the user mode or the programmer mode, all of which are freely interchangeable;

(ii) an interpreter program that processes programmer, author and user control codes and automatically switches the system from one mode to another; and

(iii) routines that automatically terminate and reschedule ADEPT for a restart later, transfer control to the operating system for processing and execution of a new job (assembly, compilation, simulation, etc.) and restart the ADEPT job when this job is completed.

For ease of transfer between machines, ADEPT was written in 37 FORTRAN IV subroutines with only two assembly-language subroutines.

2.1.4.11 CG-1. CG-1 is a course generating system (Meadow et al., 1968) which produces a course program as the result of a conversation carried on in natural language. The generated programs are in PL/I Interactive Dialect (PL/I ID) which makes use of a limited set of PL/I statements, several subroutines which perform instruction-related functions, and some syntactic rules governing the writing of programs. This dialect may be compiled and run on most IBM System/360 computers. The system has the following features and limitations:

(i) little programming skill is required by course authors, but those authors with programming skill may enter PL/I statements of their own into the generated instruction program;

(ii) only multiple-choice questions can be generated;

(iii) an instructor may insert PL/I language statements to analyse or process responses or process other data; and

(iv) the generated course allows three unrecognisable responses, after which the student is automatically cut off from the computer.

A second system, CG-2, is under development. This is an updated version

of CG-1 and enlarges the number of possible answer types and corresponding response analysis and allows more complex decision branching.

2.1.4.12 WSUCAI. The Washington State University Computer-Assisted Instruction system (Kerr et al., 1969) is implemented on an IBM 360 model 67 with 756 K bytes of core storage, an IBM 2314 disk storage device and an IBM 2321 data cell. The operating system is Multiprogramming with a Fixed number of Tasks (MFT) using the Houston Automatic Spooling Priority system (HASP). The five partitions usually run are HASP, batch processing, plotter, graphics and teleprocessing. WSUCAI uses a package written for terminal interface that runs in the teleprocessing partition. Other application programs run in that partition and no time-sharing is involved. The control program for WSUCAI includes:

(i) an instructional system, which may take either the form of successive frame presentation or review mode;

(ii) a recording system, which collects information on the sequence of frames and total instruction time by student, total operation time and frequency of operation by terminal, and frequency of each response and average reaction time by frame number; and

(iii) a management system to generate reports for instructors and descriptive system reports for those developing WSUCAI.

The control program is written in FORTRAN IV and is as machine independent as possible.

2.1.4.13 INFORM. INFORM at Philco-Ford (1970a) is a magnetic tape oriented system implemented on a Philco-Ford Model 102 Processor and a Model 173 Magnetic Core Memory of 32768 words. The flow of information through the system and the many varied forms taken by the data is as follows. Specially designed coding forms are used by the author in the INFORM Author Language and contain his step-by-step teaching techniques and curriculum material. A keypunch operator punches the information

into cards which are then recorded on magnetic tape and processed by the INFORM Translater Program. The translater generates an intermediate language and records it on magnetic tape. At the same time a "curriculum edit" listing is printed. The author can look at this listing to check the original information that he intended for his instructional material and the Author Language tape may be stored for future use. The Interpreter Program transmits the course information to the student via an individual SAVI (Student Audio-Visual Interface) display screen. The student is able to respond using a keyboard and light-pen.

The INFORM Control Instructions provide a variety of initiation and update service functions. These instructions are used to prepare curriculum files, copy curriculum programs from one magnetic tape on to another and to add, delete and replace individual cards or entire concepts during these operations.

One specific application of INFORM has been the Project GROW System (Philco-Ford, 1970b) which was designed specifically for the Philadelphia School District.

2.1.5 The need for NUTS.

One of the main reasons for designing another CAI system was that in the autumn of 1968 there was no pre-packaged system available for implementation on the IBM 360 model 67 at Newcastle, either from the manufacturer or from any other installation using CAI. We wished to combine as many of those attributes mentioned in 2.1.3 as possible in one system. Considered of great importance among these were

(i) the availability of an author language to ease the author's task of entering his material into the computer yet at the same time allowing him to provide all the student-course interactions of the type already familiar in other systems;

(ii) ability to enter courses either from a terminal or from cards;

(iii) a computational capability to be available at all times to all users; and

(iv) a recovery from malfunction such that the author does not lose any course material he may have entered or the student has to repeat as little of the dialogue as possible.

As for implementation, in an effort to design a machine independent system, FORTRAN was considered essential. It is a universal language and hence transfer of the system between machines would not require extensive reprogramming; only alterations owing to a different operating system would need to be made. Other reasons for its use were that communication of FORTRAN programs is not difficult, which would allow amendments to be made by other people, and it is quick to write for an experienced programmer and easy to debug, which would give a small elapsed time between the start of the project and the first test version, an important factor in this context.

It was not possible to include all the design considerations given in 2.1.3 and those not realised were

(i) re-entrant, shared code was not available for FORTRAN (or for assembler, even, at that time) in the operating system;

(ii) the necessity for providing an open-ended command structure was not considered important as it was hoped to provide all those facilities that would be needed, but, in any case, having chosen FORTRAN, addition of further commands would not cause any difficulty; and

(iii) a stand-alone system was designed, not a subsystem able to communicate with the operating system because a simple system was required quickly to enable further research to be carried out amd, in any event, it would be an easy matter to arrange for the command language to include functions of the general operating system by virtue of dynamic loading of these other facilities

It was in this frame of reference that NUTS was designed and implemented.

2.2 Michigan Terminal System and its influences.

2.2.1 UMMPS and MTS : A general description of the operating system.

UMMPS (University of Michigan Multi-Programming System) is a multiprogramming operating system for the IBM System /360 series of computers. UMMPS executes jobs, which are initiated and controlled from the operator's console. Each job runs in problem state and uses supervisor calls for all its input and output operations.

A job program is the basic set of instructions which are executed when a UMMPS job is run. Job programs are core resident, along with the UMMPS supervisor and subroutines. A re-entrant job program can be executed at the same time by more than one job. When a job program is written, a set of device types and a set of memory buffers of various sizes are specified. Corresponding actual devices and memory space are allocated for any job initiated with that job program, and these are retained until the termination of the job. By means of supervisor calls, jobs may obtain and release additional devices and storage space during their execution. A single device (e.g. a card reader, communications terminal, or a disk module) is available for only one job at any given instant.

The version of UMMPS in Newcastle uses the dynamic relocation hardware peculiar to the 360 model 67 in order to provide a virtual memory buffer space of 256 pages (one page = 4096 bytes) for each job. The supervisor manages real core memory with a demand paging algorithm, using an IBM Drum for secondary storage.

MTS (Michigan Terminal System) is a re-entrant job program in UMMPS. It provides the capability of loading, executing and controlling programs from remote terminals and through a batch stream. Together with UMMPS, MTS provides a simple but powerful time-shared computer system, whose salient features are these (University of Michigan, 1967).

(i) Several dozen commands are available to cause the running and monitoring of programs, the manipulation of line files, and other

communication with the system.

(ii) A system of information organised in units of lines (1 to 256 characters) and files (0 to many thousands of lines) is provided for the storage of programs and data. A file may be public or private, and a private file may be permanent or temporary. These files reside on direct-access storage devices.

(iii) When an MTS user specifies the origin or disposition of data, he may give, interchangeably, the name of a file location or a physical device. A logical device name or number is then attached to it. It may refer, for example, to a system (public) file, a new temporary private file, a card punch, or the operator's console.

(iv) A program to dynamically load programs is an integral part of MTS. It may be invoked by both commands and subroutine calls.

(v) External symbols, which have been referred to by a set of loaded programs, but not defined, may be resolved by reference to a private or a system library, which is a file containing object programs in a special format. Facilities exist in MTS and the Loader to pass over a library and selectively load only the required subroutines (and the subroutines that they need, etc.).

(vi) The MTS system makes available the IBM System /360 F-level assembler, the IBM FORTRAN IV G-level compiler, the IBM ALGOL F-level compiler, the IBM PL/1 F-level compiler, WATFOR (University of Waterloo FORTRAN load and go compiler), PIL (Pittsburgh Interpretive Language), SNOBOL4 (a string manipulation language), and UMIST, a string processor based on the TRAC text-processing language. These programs reside in system files, and are executed in the same way as user programs produced by these processors. Other powerful system

features, such as the IOH/360 input-output conversion subroutines, macro libraries, plotting routines, etc., reside in the library and other system files.

2.2.2 The dependence of NUTS upon MTS.

Although one of the main design considerations of NUTS was that it should be as independent of MTS as possible, there are some areas in which dependence upon the operating system was unavoidable. However, in each case, very little, if any, re-programming would have to be carried out if NUTS were to be implemented under a similar general-purpose time-sharing system. Those areas are described in this section.

2.2.2.1 Execution. MTS treats NUTS as though it were just another problem program. The object code is owned by the system designer but any registered NUTS user may request its use and he receives a complete copy of the system in his virtual memory.

2.2.2.2 MTS command generation. Each user initiates NUTS by transferring MTS control to his file called "nuts" which at that time contains the MTS commands to load and start NUTS. Depending upon which command he then requests, it may cause the command language processor to write appropriate MTS commands in the next available line in "nuts". On the completion of the command language processor's scan, NUTS execution is interrupted and MTS control passes to the next available line in file "nuts", which should contain the corresponding MTS commands to the user's NUTS request. Less than half of the 17 NUTS commands require MTS command generation. The commands which are produced concern the creation and copying of files and restarting with a different file assigned to a particular FORTRAN logical device number.

2.2.2.3 MTS supervisor calls. There exists two assembly language subroutines, each of which are essentially just a supervisor call. The first one is used to measure the elapsed time of students' responses and the total session time whilst the second indicates whether the user is currently using NUTS

on terminal or in batch. This is important as certain commands are only available from a terminal and within the language processors prompting does not occur in batch.

2.2.2.4 Use of permitted files. MTS allows one user to read the files of a second user provided the second user has permitted him access. This facility is used in the situation that after an author has checked out a lesson in a course he releases that lesson for general use, or, in other words, he permits its use. From that point on, a student may use that lesson, but, of course, only whilst the author keeps it released.

2.2.2.5 Privileged use. If a user in MTS is afforded priviledged status then he may look at other people's files without their consent. Thus, every author is given this status as from time to time he may wish to observe how some of the students are progressing through one of his courses. He does this by looking into their response file for that course. In another instance, authors may wish to look in the NUTS student index and lesson index to see which students are currently using NUTS and which lessons are available. These files are owned by the system designer but are not permitted by him as they are updated periodically and permitted files may not be written into.

2.3 A glossary of terms for NUTS.

author — a user who is able to create and build pieces of instructional material, extensively check them out before releasing them for general use, then monitor the performance of the subject using this material.

author language — this is used by authors only, to write courses of instructional material. It is especially designed for this purpose and so includes source statements which specify material to be displayed and acceptable student responses, access past student performance information, load and unload return address stacks, etc. The author language processor analyses the source statements and translates them into intermediate code for future calling via a controller. If errors are detected, appropriate diagnostic error messages are produced.

calculating language — this is used by authors and students alike during a NUTS session to write programs for applications that involve mathematical computations and other manipulation of numerical data. The calculating language processor analyses the source statements and translates them into intermediate code for future calling via a controller. If errors are detected, appropriate diagnostic error messages are produced.

command language — this is the principal medium of communication between NUTS and the various users of the system. The author may employ the facilities of the command language to construct, check out and release his lessons, to monitor his own and his students' response files, and to use the calculation facilities. The student is able to use released courses and invoke the calculation aids.

course — the name given to a collection of from one to ten lessons which will normally be subject interdependent with one another and together form a completely self-contained study. The actual name of a course is from one to five upper case letters.

examples : MATHS, HIST. invalid : CHEM2, PHYSICS.

lesson — the name given to a piece of instructional material. The author prepares this by coding statements in the author language. The name of a lesson may be from one to five upper case letters followed by a digit. The letter part of the name must be identified to the name of the course of which the lesson is to be a part. Lesson Ø of a course, however, is of special interest. It must contain the beginning of a course and, for that reason, must be the first lesson of a course to be created and the last lesson of a course to be destroyed.

examples : MATHS2, PL1. invalid : MATHS, PHYSICS, PL12.

PIL — The Pittsburgh Interpretive Language is a remote terminal language designed to provide the user with much assistance through the use of terminal diagnostics, user interaction with the machine and associated error recovery procedures. It is available to users during a NUTS session when they may wish to employ a facility for numerical and string manipulation (Flanigan, 1968).

program — a set of statements from the calculating language which the user writes to carry out a mathematical computation for him during a NUTS session. The name of a program may be from one to six upper case letters.

examples : CUBICS, SQ. invalid : QUARTIC, ORDER2.

response file (student) — for every course in which he takes part, a student is assigned a response file. In this is stored his progress through the course, that is, the specific route he has taken, the responses he has made and whether they were expected or unanticipated, and how long he took over each response. The information is available to the author to decide any future branching in the course. Also contained in the response file is the restart address (and other allied information) from where the student recommences after a voluntary or enforced interruption of the course. The name of a student response file is the associated course name followed by the characters "#Ø". examples : MATHS#Ø, HIST#Ø. invalid : MATHS#2, HLSTØ.

response file (author) — for checkout purposes, the author may use up to ten different response files per course. When he checks out a course, he is given the option of which response file to use. If he chooses one which has already been used he then has the option to recommence from the stored restart address, or specify from where in the course he wishes to proceed. The name of an author response file is the associated course name followed by the character "#" then a digit. examples : MATHS#7, HIST#Ø. invalid : MATHS#23, CUBICS#3.

segment — each lesson in a course is divided into segments of instruction material. These usually comprise the smallest piece of material, the subject content of which embraces one specific concept. The start of a segment, therefore, is intended to be used as a possible restart point in a course. Whenever a segment start is encountered during a course, this address is entered into the response file along with other such information as the state of all the

variables and stacks, all of which is necessary to represent the current state of the student's course. There may be up to 99 segments in any lesson of a course. Their name is the letter "S" followed by an unsigned integer from 1 to 99.

example : S2, S73. invalid : S123, SØ.

student — a user who is the subject of the instructional material (at certain times one author may be acting as a student to another author). Unlike the author, he has no freedom to move around the course at will, unless the author gives him the option of doing so. However, he has available to him certain calculation aids and the housekeeping associated with them.

userid — every student who will use any of the available courses must first be registered. This entails entering his userid (user identification), which is also his MTS userid, in a NUTS catalogue of students.

2.4 Description of files used in NUTS.

Despite the fact that all files reside on disk, a direct access storage device, some are used in a sequential manner. However, the best classification of files is with respect to ownership.

2.4.1 Files owned by the designer.

The system designer owns all the system files. These comprise the object modules, the student index and the lesson index. He is not a user of the system but acts in the capacity of manager. The object modules are contained in sequential files and are permitted for general use. The student index contains an index of the MTS userid of all students and authors joined to NUTS. It is updated by the manager by hand and searched whenever an author issues a request to view the response file of one of his students so that the system first ascertains whether the student is in fact joined to NUTS. The lesson index contains an index of all the lessons, together with their author, currently released. It is updated automatically whenever an author releases a lesson and whenever the author either restricts further use of the lesson or destroys it altogether. The index is searched whenever a student issues a request to use a course so that the system can first find out if the appropriate lesson is available and then, if it is, determine the unique name of the file which is to be used.

2.4.2 Files essential to all users.

When the user is joined to NUTS, the manager creates and initialises two files for him. As mentioned in 2.2.2.2, every user owns an MTS command generation file. It is initially created with three lines only and every time a user wishes to invoke NUTS he transfers control to the first three lines which load and start the system. Any subsequent command he requests may generate an MTS command but these are written sequentially in the command generation file commencing at line 4. The other file is the user's file catalogue. Nearly every NUTS command operates on one type of file or

another. Also, some commands are only available to authors. Consequently, every user is given a file catalogue which stores such information as whether the owner is an author or a student, what files he possesses, what the limits are on the number of files of each type he may possess at one time and how much time he has used in NUTS. Whenever the user needs access to a file, his catalogue is searched. For ease of storing file names, a unique numeric code is used. This is based on mapping the letters of the alphabet into the numbers 1 to 26 and then converting any sequence of letters as though it were a base 27 number system, neglecting Ø.

2.4.3 Files exclusive to authors.

Only authors may own lesson files. This file contains instructional material which the author has generated using author language statements. Both the source statements and the corresponding intermediate code are stored in this file so that it is the same file that is used firstly by the author in the development stage and then, after release, by the student in the instructional stage. Its organisation is as follows. As the author enters his lesson, each of his source statements is given a line number. The lesson file has a line directory to denote whether or not any line out of the total possible is contained in the lesson. Source lines are stored with their length preceding them. The placement of a source line in the lesson file is given by a convenient algorithm which utilises the direct access capability available. For convenience at translation time, the currently largest line number and the line number of the END source statement are stored. The intermediate code is stored sequentially in blocks of ten lines and a counter keeps note of the number of such blocks produced. Also stored are the addresses in the intermediate code corresponding to the start of each segment used.

2.4.4 Files owned by any user.

Program files, a PIL statement file and response files may be owned by authors and students. Using the calculating language, authors and

students may construct and store programs for their own use at any time. The organisation of program files is almost identical to that of lesson files. The only differences are the size of the line directory since the range of possible line numbers is smaller and the absence of a segment start address directory since programs are not divided into segments. PIL statements written during one session with PIL are stored in the PIL statement file for use in subsequent PIL sessions. This file may not be destroyed by a user. It is created when the user is joined to the system and, of course, may never be used if not desired. A response file stores a complete record of a user's interaction with a course. Students create a response file implicitly whenever they commence a course and may only have this one response file for that particular course. Authors, however, because they need to check out the constituent lessons within a course before release, are given the option to create up to ten response files per course. Within the response file is kept such information as the exact route through a course, the responses given to each question attempted and the time taken for the response. This information is stored for the author's benefit as he may base his strategy for course sequence on it. The total time to date for the course and that of the last session is also kept.

2.5 The command language : design and implementation.

2.5.1 Introduction.

The command language can be used in two modes called conversational and non-conversational. In conversational mode, the user remains on-line to NUTS, engaging in a dialogue with it. In non-conversational mode, the command language serves as the job control language for operations that do not require a dialogue with the user, that is, operation submitted to the system for execution without user-monitoring.

2.5.2 The choice of commands.

The rationale for deciding which commands should be available to NUTS users was as follows.

2.5.2.1 Processors. Commands were needed to invoke each of the four processors available within NUTS. Consequently, choice of command in this case is equivalent to choosing that particular processor. The fundamental processor in a CAI system is that for the author language. Hence, the BUILD command was designed to call upon the author language translater. In order to give students a simple calculation facility which they could use at any time within NUTS - even during a course - a simple programming language was included. This enabled students to develop and test programs, store them away, then return to them at some later stage. The CALC command places the calculating language translater at their disposal. At a later point in time in the development of NUTS facilities, it was thought that perhaps a language which students may have learned before ever using NUTS, a language whi h would be available at any time in MTS, should be available to them during a NUTS session. The language that fitted these requirements was PIL (Flanigan, 1968), the Pittsburgh Interpretive Language. The command to request PIL is simply PIL. In order to provide a simple desk calculating device, a desk machine, activated by the command DESK, was included. This, of course, has now been superseded by the direct mode facilities of PIL.

2.5.2.2 Execution. Having developed lessons and programs, users need commands to execute these. To allow authors to check out their lessons, and students to enter into courses, there is the COURSE command. The CALC command, which has already been mentioned, invokes the controller after a successful translation. However, for execution of correct programs without re-translation, there is the PROG command.

2.5.2.3 General file handling. So that users are able to update files from time to time, three commands were designed for this purpose. Firstly, there is the INSERT command, which allows users to insert, replace or delete lines in a file by referring to the actual line number. Secondly, there is the COPY command, which allows the wholesale copying of one file into another, instead of re-typing. Finally, the RID command destroys any unwanted file.

2.5.2.4 Housekeeping. Two commands were devised to aid the user in his housekeeping activities. The CAT command effectively gives a listing of the file catalogue. In the case of an author, he is told of all the files he possesses, that is, lessons, programs and response files, but the student only finds out which programs he owns, for his response files are not available for his inspection and, of course, he does not own any lessons. Having found out which lesson and program files he owns, the user may obtain a listing, either in part or in full, by using the LIST command.

2.5.2.5 Author's file handling. By virtue of his privilege of owning lesson files, the author has available to him the following commands. To obtain a list of all currently released lessons and their owners, there is the LESSON command. To release a lesson of his own after careful check-out, there is the REL command. Then, quite naturally, there is the opposite command, RES, which restricts use of a lesson until the author has been able to modify and re-release it. The response file contains much information which the author may wish to see from time to time. Firstly,

when he is checking out his lesson, he uses the RFILE command to give him information about one of his own response files. Then, after releasing the lesson and when the students are using the course, he may use the SFILE command to check the students' response files. This can be done while students are engaged on the course.

2.5.2.6 NUTS initiation and termination. To invoke the system, the MTS command, "/SOURCE NUTS", is used. When a session is over, QUIT returns control to MTS.

2.5.3 Command mode.

In reply to the initiating MTS command and in general when it is ready to accept the next command, NUTS signifies that the user is in command mode by prompting with an asterisk, then unlocking the keyboard after a carriage return-line feed. In conversational mode, the user's contribution to the dialogue consists of the commands and source language statements, if any, that he enters during the execution of his task, and the replies he makes to the messages issued by the system. The system's contribution consists of the messages it issues to the user, the responses it makes to his commands, and the requests for the next command. During execution of a non-conversational task, there is no communication between the user and the system. The system analyses each command of the command sequence and, if it is valid, executes it. If the command is invalid, the system ignores it and continues until a valid command is read.

2.5.4 General description of the command language processor.

Every command entered by the user is executed interpretively. The command is first read into a buffer and all embedded blanks removed. The leading string of alphabetic characters is then converted to a unique numerical value using the base 27 number system technique. This value is compared with the table of commands. As the number of commands is fixed and small, hash table techniques were not employed. Simple table lookup provides a quick enough check. If a match is not found, an error is assumed and the command prompt re-appears. If a command is recognised,

further syntax checking is carried out on its operands. Except for QUIT, if a valid sequence is present, the command is executed by means of an appropriate subroutine call, but in some cases this follows the writing of the necessary MTS commands in the MTS command generation file. When QUIT is recognised, the session time and total time to date are printed and stored, after which time control returns to MTS.

2.5.5 Command implementation.

Nearly half the NUTS commands use the MTS command generation technique. This was necessary for two main reasons. Firstly, FORTRAN execution under MTS requires that files needed during the run be attached to the appropriate logical device number at load time or when execution is restarted after a pause in execution. As the particular file is only determined from analysis of the NUTS command, allocation of the file must therefore take place together with a restart. Secondly, some commands require that files be created. This may only be carried out in MTS command mode so that a pause in execution of NUTS is required. To satisfy these two conditions, NUTS execution is temporarily stopped using a FORTRAN PAUSE statement, but prior to this the necessary MTS commands have been written into the command generation file. Control passes to MTS with the commands coming from the appropriate lines in this file. The last command is always a restart so that NUTS execution recommences.

The following table contains a list of all the NUTS commands. For each command, information is given about its purpose, when it is available and to whom, its modes of use and which MTS commands are generated. More detailed discussion of implementation follows in the rest of this section.

COMMAND NAME	PURPOSE	WHEN AVAILABLE	MODES OF USE	MTS COMMANDS GENERATED
BUILD	to invoke the author language translater	authors only, terminal and batch	1. prompted entry of statements then translation 2. modifications then translation 3. translation only	1. CREATE lesson file, if new 2. RESTART with lesson file
CALC	to invoke the calculating language processor	authors and students, terminal and batch	1. prompted entry of statements then translation 2. modifications then translation 3. translation only	1. CREATE program file, if new 2. RESTART with program file
CAT	to inform the user which files he possesses	authors and students, terminal and batch	1. student is told only which programs he owns 2. author may request lessons, programs or response files or all three together	
COPY	to copy an existing file into another file	authors and students, terminal and batch		1. CREATE file, if new 2. COPY exactly
COURSE	to take part in a course	authors and students, terminal only	1. student restarts from where he left off 2. author may choose which response file and the restart point	1. CREATE response file, if necessary 2. RESTART with response file and appropriate lesson
DESK	to invoke the sequence controlled desk calculator	authors and students, terminal and batch		

COMMAND NAME	PURPOSE	WHEN AVAILABLE	MODES OF USE	MTS COMMANDS GENERATED
INSERT	to edit program or lesson files without re-translation	authors and students, terminal and batch	1. prompted entry 2. entry with line number included	1. CREATE file, if new 2. RESTART with file
LESSON	to obtain a list of released lessons and their authors	authors only, terminal and batch		
LIST	to obtain a current listing of a file	authors and students, terminal and batch	1. the complete file 2. from a starting line to the end 3. both starting and finishing lines specified	1. RESTART with file
PIL	to invoke the PIL interpreter	authors and students, terminal and batch		
PROG	to execute a successfully translated program	authors and students, terminal and batch		1. RESTART with program file
QUIT	to terminate a NUTS session	authors and students, terminal and batch		
REL	to release a lesson for general use	authors only, terminal only		
RES	to restrict the use of a lesson	authors only, terminal only		
RFILE	to give the author a listing of one of his response files	authors only, terminal and batch		1. RESTART with the response file

COMMAND NAME	PURPOSE	WHEN AVAILABLE	MODES OF USE	MTS COMMANDS GENERATED
RID	to destroy a file	authors and students, terminal only	1. a student may destroy only programs 2. an author may destroy any of his files	
SFILE	to give the author a listing of one of his students' response files	authors only, terminal and batch		1. RESTART with the student's file catalogue and his response file

2.5.5.1 BUILD. The first operand, the lesson name, is converted to a unique numerical value. The digit part of the name is the next character after the string of letters. A table lookup takes place in the author's file catalogue to see whether the lesson exists, whether lesson ø exists and whether the lesson is currently released or not. For protection purposes, two constraints are made. Firstly, a released lesson may not be re-translated. It must be withdrawn from general use before the author may update it. Secondly, lesson ø must be the first lesson created in a course. This is because lesson ø must contain the starting point of that cause. After the restart, a subroutine transfers control to the author language translater, passing across the mode of use, the starting line number and the increment as parameters.

2.5.5.2 CALC. The first operand expected is the program name, for which a unique numerical code is derived. A table lookup into the user's file catalogue ascertains whether the program already exists. A subroutine similar to that for BUILD is called after the restart.

2.5.5.3 CAT. After the mode of use is analysed, a subroutine is called which searches the appropriate part of the user's file catalogue and decodes each entry from the unique numerical code back to the actual file name.

2.5.5.4 COPY. The expected operands are two file names, either lessons or programs, separated by a comma. A lesson name is easily recognisable by its trailing digit. Table lookup on both files follows to see whether they exist and what type of file they are. As well as the obvious restriction that the first file must already exist, two further constraints are imposed for protection purposes. Firstly, both files must be of the same type. This is for the simple reason that certain statements are available in one language but not in the other. Secondly, as mentioned before, if a new lesson must be created, then lesson ø of that course must already exist. No subroutine is needed, only the MTS command to give an exact copy. However,

only the source statements are copied, so that retranslation is essential before execution of the new copy.

2.5.5.5 COURSE. The command language interpreter first recognises the course name.

For an author, table lookup occurs to see whether lesson ø exists, that is, whether the author is going to run one of his own courses. If he is not, he is treated like a student. The author is then asked to indicate which response file he wants to use. A single digit is sufficient to identify it. Table lookup determines whether or not it exists. For a new response file, that part of the file corresponding to lesson ø is initialised by a subroutine call. This entails reading from lesson ø the number of real and integer variables needed and setting up storage for them. Also, the return address stacks corresponding to lesson ø are initialised, and the course restart address is set to the beginning of lesson ó. The course restart address is then read from the response file and the author is given the choice of continuing from that segment or specifying a different segment or lesson even. In any event, the required segment is determined but in the latter case a search is made in the specified lesson's segment directory to see whether the segment does exist. The actual address in the intermediate code can then be read from the lesson's segment directory. This method allows for considerable changes to be made to a lesson, but so long as the segment directory is up to date, the correct address in the intermediate code is found. A subroutine call invokes the course controller.

A student is not given the choices afforded an author. Once it has been confirmed that the course is available by a search through the lesson directory, a table lookup determines if the response file already exists. If it does not, initialisation takes place the same as for an author but in any case the course restart address is read from the response file, the actual intermediate code address determined and the course controller then called.

2.5.5.6 DESK. This command has no operands. A subroutine call places the sequence controlled desk machine at the user's disposal.

2.5.5.7 INSERT. The first operand expected is a file name, either a program or a lesson. The difference is easily distinguishable by the presence of a digit to denote the lesson number. A table lookup takes place to determine such facts as whether the file exists and, if a lesson, whether it has been released. Upon the restart, a subroutine call causes execution of the command. The mode of use, starting line and increment, and a line number limit, which, in effect, just indicates whether a lesson or a program is being updated, are passed as parameters. There are two modes of use of the command. One allows prompted entry of statements whereas the other requires specification of the line number by the user each time.

For the first mode of use, the user is prompted by a line number each time. The length of the input line is calculated using the right-most non-blank character. Using a simple relationship, the MTS line number corresponding to the NUTS file line number is evaluated, and, using a direct access output statement, the line is written to disk. Knowing the length conserves disk space. The line directory is also updated. A further prompt is then issued. A blank input line indicates command termination. If the first character is "%", then the user wishes to override the line number and his line contains a line number followed by the contents but separated by a comma. This kind of input line is treated as though it were of the second mode, but upon completion of the output to disk, further line number prompting occurs.

In the second mode of use, a prompt occurs but without line number. The user enters his source line consisting of line number and contents, separated by a comma. The digits he entered at the beginning of the line are converted to an unsigned integer and the separator is tested. If the line number is in the correct range, the length of the actual

input line is calculated. This line is then output to disk and a further prompt occurs. As before, a blank input line indicates command termination. It should be noted that the second mode must be used to delete lines.

Whenever a previously successfully translated program or lesson is updated, the intermediate code becomes inaccessible so that retranslation is necessary before use. This guards against possible confusion which might arise if the source code in a file did not correspond to the intermediate code.

In non-conversational mode, the user receives a printed copy of each of his input lines so that he may check them afterwards.

2.5.5.8 LESSON. The command interpreter calls a subroutine which searches the NUTS lesson index file and prints out those lessons and their authors which appear.

2.5.5.9 LIST. The first operand expected is either a lesson name or a program name. Table lookup of the user's file catalogue occurs to determine whether the file exists. The mode of use is determined by a search for a blank, for all the file, an unsigned integer within parentheses, for the rest of the file commencing at that line number, or two unsigned integers separated by a comma and enclosed within parentheses, for that part of the file between the given line numbers. The line numbers are checked for magnitude.

A subroutine is called when NUTS restarts to list those lines indicated. To save unnecessary searching of the line directory if the user has specified a finishing line number greater than the last line number in the file, the current last line number value is read from the file. The storage of the length of the source line saves time on output to a terminal, which is very slow.

2.5.5.10 PIL. This command has no operand. When the command interpreter recognises it, a subroutine is called. In this, a branch is made to the PIL interpreter but first a dummy input stream is used. This loads into the PIL workspace the statements that have been stored in the user's PIL

file since his last usage of PIL. When the user indicates his desire to return to command mode, a dummy output stream first issues the requests to empty the PIL file, then copy the contents of the current PIL workspace into the file. After this has been accomplished, the return request is properly furnished and control returns from the PIL interpreter to NUTS.

2.5.5.11 PROG. The only operand for this command is the program name. A table lookup occurs in the usual way to see whether the program exists. Then, a subroutine call invokes the calculating language controller.

2.5.5.12 QUIT. This command has no operand. When it has been recognised, a supervisor call within an assembly language subroutine is made to determine the time. From this and the result of a similar call when NUTS was initiated, it is possible to calculate the length of the session. The total time to date is then read from the user's file catalogue, and the updated total and session time are written back to it. In addition, a copy is written out to the user, after which control returns to MTS.

2.5.5.13 REL. The operand is a lesson file. A table lookup on the author's catalogue file occurs to find out if it exists. On top of this, a further lookup occurs on the corresponding lesson Ø to determine if it has been released. Lesson Ø must be the first lesson released in a course as it contains the starting point for that course. If the author is trying to release lesson Ø, a table lookup occurs on the lesson index file. This is to find out if the course name is unique. To avoid possible confusion, authors are not allowed to have courses of the same name. The solution is for the second author to rename his course by copying the lessons into other lessons with a different name. All these conditions being satisfied, the course name, lesson number and author's userid are entered in the lesson index.

2.5.5.14 RES. As in the REL command, the interpreter expects a lesson file. A table lookup on the authors file catalogue determines whether the lesson exists. If it does, a further lookup in the lesson index determines the state of the lesson at that time. If the lesson is released,

a bit in the table entry is set to indicate the restricted condition. This was done in preference to deletion of the entry as it was expected that restricted lessons would be released later.

2.5.5.15 RFILE. This command has one operand, a response file name, which the interpreter easily recognises by the character "#" and a digit after a course name. A table lookup then determines whether the response file exists. The contents of the response file are displayed in the following way. The first line gives the current position in terms of lesson number and segment number, both of which are read from the same line of the response file. The route through the course is then read into a buffer and from this information one line is generated in sequence for each question attempted. The route is stored as a sequence of numbers, the values of which are the lesson number multiplied by 100 with the question number added. As up to 99 questions per lesson are allowed, the mapping is 1-1. The information corresponding to each attempt is stored in direct access fashion, the MTS line number consisting of the route number, defined above, as the integer part, and the attempt number of the question, which allows for up to 999 entries, as the fraction part. Working through the route buffer in sequence and counting which attempt number per question is required indicates which line of the response file is to be accessed. This line of information is read in and the following format is given to the author. He receives the lesson number, question number, a code for either anticipated answer, unanticipated answer or not answered, truth values for the response elements if an anticipated answer, time taken, and, finally, the actual reply if unanticipated. He receives one such line for every question attempt. Finally, the total time to date and that for the last session with the course are given.

2.5.5.16 RID. This command expects a file name as its operand. An author may destroy a lesson, program or response file whereas a student may only get rid of a program. A table lookup takes place to determine if the file exists. For an existing program or response file all that

happens is that the appropriate entry in the user's file catalogue is deleted. A further constraint, however, is placed on a lesson if that lesson is lesson Ø. The table lookup also indicates whether any other lessons exist. If one does, then lesson Ø is not destroyed, for it must be the last lesson in a course to be destroyed as it contains the course's starting point. If the lesson can be destroyed, then a table lookup occurs in the lesson index file to see if an entry appears there. If one does, it is deleted. Once the table entries have been removed, the file is removed from MTS.

2.5.5.17 SFILE. This command has a course name as the first operand. Table lookup for lesson Ø occurs to determine whether the author owns such a course. If he does, the command interpreter then determines the userid of the student whose response file is to be listed. Table lookup in the student index then shows if that particular student has been joined to NUTS. To determine whether the student has yet commenced the course, his file catalogue is searched to see if the particular response file exists. A subroutine call similar to that for RFILE is then made.

2.6 The author language : design and implementation.

2.6.1 Previous author languages : the need to create another.

2.6.1.1 Author language attributes. An author is unlikely to have programming skills or an assistant to code instructions for a computer. In fact, the author in a CAI system should be able to write in his own language with the minimum of restrictions the instructional material he plans to use. He will also wish to employ the computer logic to provide individualised instruction, and it should be easy for him to do so using instructional strategies with which he is comfortable.

An essential characteristic of an author language is that it be user oriented without denying the author access to any of the system capabilities. For example, the novice should be able to prepare material for instruction after only a short time studying the language, and the experienced author should be able to use the full capacity of the computer to construct as complex a procedure as he wishes.

Zinn (1967) suggests that CAI systems should provide author input facility by author languages at three levels. In the first, the author only enters his text and rules for evaluating answers in some standard pattern of instruction, e.g. a PLATO tutorial teaching logic. No knowledge of computers is required. The second level allows the author to specify his particular pattern of instruction in a relatively simple language that can be learnt in a short time. At the third level, an author having some training in computer programming extends the author language by writing out his own routines and strategies, employing the full capability of the computer system by perhaps using machine code.

A comprehensive list of important constituents of an author language is given by Adams (1969). For variables, he suggests several dozen arithmetic variables, including floating point, at least 100 logicals, probably more than 10 alphameric strings each of about 100 characters, data structures having programmer specified format and lists. Control statements might include : conditional, iteration, transfer, subroutine,

storage allocation, restart point, time dependent transfer and log entry. A macro processor within the author language and library routines written in other languages which can be used in the author language would be desirable. Important processing operations on natural language strings include : definition or detection of substrings, transformational processing and tests on numbers or strings which might include character match, numerical equality, numerical range match, degree of match, pattern tests and general user defined tests.

2.6.1.2 Comparison of author languages. About 30 different languages and dialects have been developed especially for programming conversational instruction. This number is changing so rapidly and up-to-date documentation is so sparse that a complete appraisal of the many languages is impossible. Any comparisons which are made between languages naturally become invalidated quite rapidly by subsequent revisions of the languages.

When evaluating the merits of author languages, Frye (1968) suggests that the aspects to be considered should include : user orientation, lesson handling, record handling, conditional branching, answer matching service routines, calculation provisions and communication devices. He points out the two methods most often used when comparing languages. These are to categorise their capabilities, noting the absence of certain features and to code a sample instruction sequence in each competing language, noting some efficiency measure such as the number of lines of instructions for each task. Among the pitfalls in these comparisons are :

(i) the language documents are not equally current;

(ii) the categories on which the comparisons were based were taken from one of the languages, introducing a bias; and

(iii) the test cases were selected from those particularly suited to one of the languages.

A comparison of two standard languages for authors with two low-cost languages, one an author language, the other an extended scientific language

is given by Zinn (1968).

2.6.1.3 Types of author language. Zinn (1968, 1970) has suggested classes into which author languages may be grouped, depending on such factors as modes of use, capabilities and development. Here, however, we discuss a classification into four types :

TYPE 1 - presentation of successive frames,

TYPE 2 - conversation within a limited context,

TYPE 3 - presentation of a curriculum file by a standard procedure, and

TYPE 4 - interactive problem-solving languages.

TYPE 1 has evolved because the most common application of computers for instruction appears to be an extension of programmed instruction. Most languages serve this function and are characterised by their convenience for displaying text, acceptance and classification of relatively short strings of texts in the student's response, automatic recording of performance data and implicit branching determined by the categorisation of an answer or the contents of a counter which is part of the response history.

Conversation within a limited context, TYPE 2, is offered by only a small proportion of computer-based instruction programs of the tutorial variety. These encourage additional initiative on the part of the student and provide a meaningful reply whatever he may do. Typically, the author must provide in the instructional program a set of conditional statements which, for any stage of discussion, make the computer reply dependent not only on the current response, but also on the history of the conversation.

Some languages are suitable for writing strategies which can be applied to various files of content, TYPE 3. The author adopts a logic which can be defined in a procedure statement. The programming of the logic may have been done by the author or by a programmer experienced in using the system. The author applies this and other strategies to

curriculum (or data) files of indefinite size.

Languages for on-line programming and debugging of simple problems, and which also have some string processing capability, TYPE 4, can serve an author as well or better than so-called author languages. However, interactive programming on a general-purpose system is not likely to include the proctor operations and other systems support which may be important in educational investigations. Such features could be added; the cost of the modifications depends on the characteristics of the operating system. Some general-purpose systems already have convenient file handling routines, protect and permit procedures, etc. Through suitable modifications, some of these languages and systems could serve most of the needs of instructional applications.

A discussion of languages of these latter two types, complete with summaries of six examples and their more useful procedural features, is given by Lyon and Zinn (1970).

2.6.1.4 Author languages currently in use. TYPE 1.

COURSEWRITER I (Mayer, 1964) was one of the first languages used and was devised originally for use on the IBM 1440 system. COURSEWRITER II (IBM, 1968) is similar except that it has additional operation codes for controlling visual displays, using macros, calling user-defined functions and making use of strings, counters and switches to control course execution based on student performance. Its main features include :

(i) the ability to cause a pause in execution for a set period of time;

(ii) a provision to interrupt the student before he finishes his response;

(iii) user-defined label fields;

(iv) a macro facility to prevent excessive repeated coding; and

(v) user-written functions, coded in assembler, to make the language flexible and open-ended.

However, COURSEWRITER II is inadequate with respect to the recording and manipulation of data. Records are limited to a small number of strings, counters and switches and calculations are restricted to integer arithmetic

on two counters only. A calculational capability would also be desirable. Though easy to learn, the implicit branching conventions may cause some authors a little concern.

CAL (Course Author Language) at Irvine (Keller, 1968) generates programs which are organised into courses, chapters, sections and lines. Automatic line numbering is given by the system but may be over-ridden by the author. Program statements may extend beyond one line and may have labels in addition to line numbers. Desirable features include :

(i) some statements may be either executed immediately or stored as part of a program for later execution and hence the course author has available as debugging aids such statements used in constructing a course as assigning values to variables, typing out data or statements, etc;

(ii) allowable data types include both logical and string variables as well as numerical variables of both real and integer mode;

(iii) three special time counters which contain the total terminal time since the student started the course, the total terminal time since the beginning of the current session, and the time required to make a response to the last input statement, and

(iv) provision of a computational capability to students, in the context of the course program.

However, a branching facility depending upon previous student history would be an advantage.

COMPUTEST (Starkweather and Turner, 1966) is a problem-oriented language for computer-assisted instruction, testing and interviewing, designed for an IBM 1620. Sequences of instructional material and test questions may be written in natural language and a variety of prompts may be used for the recognition of a correct answer from typewriter input. The answer may determine the comment returned and the choice of next question to be asked.

PILOT, an acronym for Programmed Inquiry, Learning Or Teaching,

(Starkweather, 1968) was developed from COMPUTEST. It is written in PL/I and was designed so that its use is not restricted to a particular manufacturer's equipment. In fact, it is general enough to be used at any level of program complexity on a range of machines from a small computer with a typewriter to a large system with many typewriters and visual displays, using different courses simultaneously. Desirable features include :

(i) the speed with which a small subset may be learned and used;

(ii) a powerful subroutine facility;

(iii) provision to store comments for subsequent perusal by the author; and

(iv) simple string operators.

However, limitations exist, such as :

(i) the language becomes quite unreadable when complex programs are written;

(ii) there is no provision for numerical matching or display of numerical values;

(iii) only very elementary algebraic expressions are allowed; and

(iv) there is no student performance record.

Most author text codes in ADEPT (Engvold and Hughes, 1968a) are similar to those of COURSEWRITER for the sake of compatability. Additional codes have been added to take advantage of the display capabilities and the facility to call upon other catalogued procedures from the operating system.

LYRIC (Silvern and Silvern, 1966a) shows a remarkable resemblance to COURSEWRITER, especially that some of the operation codes introduced replace the earlier user-defined functions of COURSEWRITER such as editing of superfluous characters, specification of keyword matching, percentage matching and numerical limits. No attempt has been made to provide real variables, a calculational capability or student information records.

WRITEACOURSE (Hunt and Zosel, 1968) was designed so that the language should be natural for the teacher and its syntax and semantics should conform to his habits. Also, readability and machine independence were sought. To meet these criteria, WRITEACOURSE was modeled on ALGOL and its translation program was written in PL/I. Despite the ease with which it can be written, WRITEACOURSE lacks any response matching more sophisticated than exact matching and does not provide any record of the student's interaction. There is a limited arithmetical capability based on a set of counters but the extent to which these may be used only includes binary operations using integer arithmetic.

The majority of instructional programs written for the PLATO system (Bitzer and Easley, 1965) have used a tutorial logic programmed in an extended FORTRAN called CATO. This gives a format for linear teaching which makes the instructor's task very easy. However, recently, CATO has been used for the preparation of a high-level language called TUTOR (Avner and Tenczar, 1969) which resembles COURSEWRITER and thus allows authors to design their own strategies.

For a system which has the capability to display graphics, the author language must deal with the problem of description of spatial coordinates and diagrams on the screen. INFORM (Philco-Ford, 1970a) includes one approach to handling this task. The author prepares the display, denotes the region for a correct answer from the lightpen, etc., in the form in which it is to appear on the screen; an assistant punches this information line by line on cards and an automatic translater prepares it for interpretation by the operating system. The language's particularly good features are:

(i) the facility to pause for a specified amount of time;

(ii) when displaying text it is possible to add, insert, overlay or erase other text; and

(iii) there exist system counters for such information as correct answers, wrong answers, time-up answers, etc., which are

automatically incremented and are initialised again at the start of every unit called a topic.

However, the computational capability is very limited. Only binary operations between 32 counters, 32 switches and 8 return registers are allowed. The response analysis allows some simple keyword matching but no numerical matching at all.

DIALOG (Kristy, 1968) has a highly-structured mode for conversational entry of curriculum files into the machine. The user selects from prescribed formats, enters strings of text which are to be displayed to the student, or enters alternative answers which are to be searched for in the student's response. As increasing control is assumed by the system, the chances are improved that sufficient information for some conversation with the student will be obtained from the author. However, it does not follow that the quality of the material will be correspondingly higher.

TYPE 2.

For the express purpose of making the machine reply depend not only on the current student response but also his previous inputs, MENTOR (Feurzeig, 1965) was developed at Bolt, Beranek, and Newman, Inc. Because the history of the conversation is stored almost automatically, and complex conditional expressions can be written with considerable ease, it is convenient for describing a dialogue of this nature. MENTOR is an interpreter written in LISP, which uses a special "front end" to make LISP accept inputs that are well suited for general usage. At execution time, MENTOR does not recognise arbitrary natural-language responses, but only items from a list of strings. Any string to be output is simply prestored, not generated.

ELIZA (Hayward, 1968) was originally developed by Weizenbaum (1966, 1967) to study natural language tutorial conversations between man and machine and the importance of context to both human and machine understanding. It is less

convenient for conditional expressions but makes considerable use of list-processing routines to divide a string of characters, the student's response, into words and phrases so that the reply can be assembled from elements of the input as well as material prestored by the author. ELIZA's main features include :

(i) ability to diagnose a wide range of responses;

(ii) ability to follow a line of argument along several or alternative paths;

(iii) the student need not concern himself too much about the format of his response;

(iv) the student has some control over the conversation by means of certain commands; and

(v) the programs are easily interchangeable.

However, it is not easy to produce large quantities of ELIZA "scripts", nor is the language easily adaptable to other systems.

Perhaps the author language currently in use with the most powerful features is PLANIT (Frye et al., 1968). Chief among these are the calculational capability and the facility for criterion branching. The on-line calculational capability, CALC, allows either the author or student to perform calculations involving trignometrical functions, algebraic functions, and matrix operations. The author may request the student to compute some data within a lesson and can specify that the student's answer be compared with the results of evaluating a previously defined function. PLANIT allows the author to specify conditions for branching based on the student's performance over any portion of the lesson. Conditions for branching may include response latency on any one answer or group of answers, number of errors made on any group of questions, help received from CALC (functions used or not used), the actual path through the lesson up to that point, or any combination of the preceding four points. Other useful features of PLANIT are:

(i) calculation results may be displayed as well as text;

(ii) execution may be suspended for a specified time;

(iii) there is a provision to interrupt the student before he finishes his response;

(iv) response processing includes an exact match, keyword search, phonetic comparison, numerical limit match and formula equivalence; and

(v) a subroutine facility.

Probably the only disadvantages of the language are its frame-structure restraints and the consequent lack of readability.

FOIL (File-Oriented Interpretive Language) (Hesselbart, 1968) was developed to provide conversational lesson-writing capability for potential authors on a general-purpose, time-sharing system. The interpretive mode allows few constraints to be placed on the syntax of the language and enables immediate execution of statements entered during testing. FOIL was written in FORTRAN for speed of production and the ease it offers for revision and addition. Its main features include :

(i) the ability to type out expression values;

(ii) response processing which includes an exact match, a keyword search, percentage match, numerical limits and expression evaluation;

(iii) a subroutine facility; and

(iv) performance recording which consists of an automatic trace of the student's path through the lesson and a copy of all unrecognised responses.

However, only integer arithmetic is provided and the use of indentation for compound statements, though possibly of great use, may cause considerable trouble to inexperienced authors.

TYPE 3.

A language suitable for writing strategies which can be applied to various files of content is CATO, an extension of FORTRAN prepared for the PLATO system. System programmers prepare various teaching logics or basic strategies into which curriculum authors can place their material.

Other examples of this type of author language include Teacher-student ALGOL (TSA), used at the Institute of Mathematical Studies in the Social Sciences at Stanford; Instructional Languages 1 and 2 (ISL-1 and ISL-2) designed by RCA, Palo Alto; and SKOOLBOL, used at the Learning Research and Development Centre, University of Pittsburgh.

TYPE 4.

APL has been used to provide instructional material (Gross et al., 1969). Features to recommend its use in CAI are :

(i) the language is definitely interactive and conversational;

(ii) all input/output is unformatted via APL functions;

(iii) the use of logical operators and a random number generator together with branches to statement numbers or labelled statements allow conditional branches to be written with ease;

(iv) a wide range of mathematical operators and user-named variables give a powerful calculational capability; and

(v) a time-of-day function allows response times to be stored.

PIL (Flanigan, 1968) has been employed in a similar manner but the requirement that strings be enclosed in double quotes when used in input and output statements is not desirable.

Similar languages include Beginners' All-purpose Symbolic Instruction Code (BASIC) from Dartmouth; Conversational Algorithmic Language (CAL) from the University of California, Berkeley; Formula Calculator (FOCAL) from Digital Equipment Corporation; the Engineering and Scientific Interpreter (ESI) from Applied Data Research, Inc., and TELCOMP from Bolt, Beranek and Newman, Inc.

2.6.1.5 The need to create another author language. NUTS was to be implemented under the Michigan Terminal System which offered a wide choice of existing algebraic and symbol manipulation languages. However, none of these was considered suitable for conversational dialogue for one or more of the following reasons :

(i) some authors would be non-programmers and hence the notation would appear quite foreign to them;

(ii) program listings would not readily illustrate the structure of the dialogue owing to numerous superfluous characters;

(iii) potential authors would need to know far more of the language than appeared directly related to their material;

(iv) many desirable features, in particular performance recording, would not be available; and

(v) the frequency of revision and correction of dialogue programs demanded on-line editing and debugging.

Consequently, an author language had to be used. However, at the time of requirement, none was available from the manufacturernor was there another author language which could be easily implemented under the current operating system. In addition, although much information on the desirable characteristics of such languages was gathered from studying those languages mentioned in 2.6.1.4, none of these was considered suitable for the variety of dialogue development anticipated. The search for and development of new characteristics and the widely varying demands made by authors rendered such features as the frame structure restraints of PLANIT or the limited computational facilities and student performance recordings of COURSEWRITER II and FOIL a severe drawback. The result was the design of another author language, but one which could be easily learned and conveniently used by subject matter experts and educational technologists.

2.6.2 Elements of the language that were needed.

Bearing in mind the comments in the last section on previous author languages, this section describes the design considerations and resulting language. It is a more complete version of that given by Dowsey (1970c). A detailed description of the language itself is contained in Appendix A.

2.6.2.1 Full computational facilities. It was considered important that the author have available full computational capability as he may wish the CAI

program to demonstrate complex calculations resulting from students' inputs, for instance. Thus, real constants as well as integer constants, user-defined variable names and a wide range of standard functions are allowed. The convenience of complicated subscript expressions and standard function arguments is catered for by allowing up to five nestings of expressions and/or arguments.

Two features are included which do not exist in FORTRAN IV. Firstly, an additional operator is used for integer division, but it differs from that usually seen in ALGOL by the fact that either or both of the operands may be of mode real. Rounding occurs first, if necessary, before the integer division. Secondly, a multiple assignment statement for subscripted variables was considered advantageous. An array element appears on the left-hand side and any number of arithmetic expressions, each separated by a comma, appear on the right-hand side. Values are stored in successive elements commencing with that element specified.

2.6.2.2 Labels. In order that certain parts of a lesson be distinguishable from others and, indeed, any statement be addressable from any other, labels were included. They may be assigned in any order without affecting the order of execution of the lesson. There are three types of label.

(i) Statement labels. Any statement except a final END may be optionally labelled for reference from other statements.

(ii) Segment labels. It was desirable that the author could divide his lessons into segments so that when the student restarts a course he does so in the segment he was in when he terminated his last session. For this purpose, segment labels are inserted, and, when encountered, they cause the system to store all variable and stack values necessary to describe the current condition of that lesson.

(iii) Question labels. To form an indexing system for the storing of student response information in the response file during a course, each question is given a corresponding question label. This label may also be referred to by the author when he monitors the student's responses.

2.6.2.3 Display of material. An essential facility for an author language is that of being able to display text. However, it was also considered necessary to allow authors to output numerical values of variables, expressions, etc. This would be most helpful to display calculations, especially those for which the student has supplied either data or intermediate results.

It was also thought necessary to allow for different formats of output. To this end, an integer value is specified after a string to indicate at which column the string will commence, and three such values follow an expression. These indicate the starting column, the length of field and the number of decimal places required after the point respectively. The latter two values allow for any possible numerical format. If the number of decimal places required is zero, an integer format is given; if it is positive, a fixed point real number is given; if negative, floating point occurs. The constraints placed are that only up to seven decimal places are allowed and the length of field must accomodate the necessary sign, point or even exponent part.

In order that the student may not be flooded by a continuous stream of information from successive TYPE statements, a PAUSE statement was included which suspends execution of a lesson for as long as the student desires. When he is ready he simply presses the RETURN key to continue.

2.6.2.4 Anticipating responses. One important aspect of CAI is that an author should be able to recognise most, if not all, of the responses which the student may make. This necessitates that the author enumerate the responses he is willing to accept for each and every question. Two ways of doing this were considered. Firstly, after the response has been read in, the response processor would search for each combination of strings and numerical values that the author specified, possibly in a conditional branch statement. The only problem with this method is that the same

string or value might appear in the different combinations specified so that repetitive searching would occur. Consequently, this strategy was not accepted. Instead, the author has the power to assign combinations of strings and values to response elements before the response is accepted and then the response processor searches for each of these specified response combinations in turn before assigning the value true or false to each of the response elements that were specified. As a result, such additional information as the number of strings, whether they must be ordered or not, the number of values and whether they must be ordered or not must be given to indicate what the author requires to be an "overall match" of that response element by the student's response. There exist 20 such response elements, #CAØ, #CA1, ..., #CA9, #WAØ, #WA1, ..., #WA9, the mnemonics standing for correct answer and wrong answer, but the author may or may not use this fact.

Even after the response elements are assigned their truth values, the author may form conditional branches depending upon combinations of these values. In this way, the author is provided with a powerful anticipated response tool, upon which he may base his course strategies.

Strings and values themselves may be specified in varying degrees of accuracy as follows.

A string may be required in three possible ways. If the author requires an exact match, he simply encloses the string in single quotes. However, if he wishes to attempt to match a response which may have been spelt incorrectly, he may specify either of the following. An unsigned integer indicates that up to this maximum number of characters may be misplaced in a string of the same length from the student's response yet still provide a match. Or, if "K" is specified, a kernel match is sought. For this, the author indicates certain characters (but no blanks) from a required answer and a match will occur if the student's response contains a string of any length which has the given characters in the given order. Such a variety of string checking was thought to provide the author

with the facility to carry out any string response evaluation that he might wish to make.

A value need not simply be an absolute number but any arithmetic expression. This allows the author to specify an anticipated response in terms of any previously calculated result and/or student's input. In addition, the author may specify a second arithmetic expression which is to indicate the error interval of the numerical response. If omitted, an exact match is attempted.

2.6.2.5 Response acceptance. The statement RESP was included for the author to indicate when he wished the student to make a response. In an effort to prevent possible omissions, one constraint was decided upon when using RESP. The author must have specified at least one response element between his indication of the start of a question by means of a question label and the RESP statement.

One powerful feature that it was decided to include was that the author be allowed to specify variables into which successive numerical values from the response would be placed, if present. The author would probably use this information in either the future reference of these particular values or further calculations using them.

Three methods of specifying a variable are allowed. Two of them quite simply are the non-subscripted variable and the array element. However, a third was devised in case a large number of values was expected and it would be inconvenient to specify a correspondingly large number of variables. This entails specification of an array together with a starting position, which itself can be either a constant or a non-subscripted variable. Thus, any resulting values are stored in successive array elements starting at the position indicated. Within any RESP statement, any combination of these three methods is allowed.

If more variables are specified than values in the response, then a predetermined large positive constant is stored in the remaining variables. This enables the author to test not only for the values entered but also the

number of such values.

2.6.2.6 Conditional expressions. Conditional expressions are included to allow authors the conditional branching capability needed in CAI. The logical quantities available for such expressions are :

(i) the response elements;

(ii) values obtained using the relational operators in conjunction with expressions, variables or constants;

(iii) implicit response elements, not yet mentioned; and

(iv) the result given back from a request to the past student performance facility, #PERF.

Any of these quantities may be combined with the logical operators to give a conditional expression.

The implicit response elements are #UA, #NA and #RTn . These, the author may not assign, but they are given truth values after the response processing has taken place. #UA stands for Unanticipated Response and it becomes true if all the specified #CA's and #WA's are false. #NA means Not Answered and becomes true if the student simply hit RETURN or entered blanks only as a response. #RTn becomes true if the time for the latest response was less than or equal to n seconds.

By far the most impressive feature of the author language when compared with others is the comprehensive past student performance facility, #PERF, which enables authors to act on student performance information stored in the student's response file.

#PERF acts in a similar way to a standard function in that it is invoked merely by writing it together with an appropriate argument. There are eight kinds of past performance about which the author may enquire, all concerned with the student's record during the current lesson.

(i) qCAd - student matched Correct Answer d to question q.

(ii) qWAd - student matched Wrong Answer d to question q.

(iii) qNS - student has Not Seen question q yet.

(iv) qNA - student did Not Answer question q.

(v) qUA - student gave an Unanticipated Answer to question q.

(vi) qRTn - student answered question q in less than or equal to n seconds.

(vii) (q1,q2,...,q3-q4,...) n XX, where n is an unsigned integer and XX may be one of CA, WA, NS, NA or UA with the appropriate meaning - out of questions q1,q2,...,q3,q3+1,q3+2,...,q4,... the student satisfied the property XX at least n times. It is in this context where it matters whether the author used CA for his correct answers and WA for his wrong answers or not.

(viii) q1-q2-q3-...- the student's path through the lesson was successively questions q1,q2,q3,... If a question was attempted twice in succession, then a match will be given only if that question number was specified twice in succession.

In addition to these enquiries, it is possible to test the truth of the conjunction of any number of these eight types. Given all these facilities, the author is provided with the capability of choosing innumerable branching criteria depending upon student performance.

2.6.2.7 Branching. It was important to allow extensive branching techniques. Firstly, branching within lessons is essential and then, since a course may consist of a number of lessons, branching between lessons becomes necessary. The following statements form the basis of the branching capability:

(i) IF. Branching may be unconditional or, more likely, as has been mentioned in the previous section, conditional. For this purpose, the IF statement was designed and depending upon the truth of the corresponding boolean expression, the branch takes place or the next instruction in sequence is executed. The executable part of the IF statement need not necessarily be a branch, as any other statement is allowed except another IF.

(ii) JUMP. The most common operation in branching an author performs is that of jumping to another part of the current lesson. To this end, the JUMP statement allows branching to any type of

label.

(iii) TRANS. In order to transfer control from one lesson to another in a course, there is the TRANS statement in which the author specifies which lesson and to which segment in that lesson control is to be passed.

(iv) BACK. Having diagnosed an incorrect or partly correct response, the author may wish to receive another attempt at the same question. An implicit branch back to the most recently executed RESP statement was considered the best way of allowing this. Hence the BACK statement was designed. Without parameters, only the branch occurs. However, all the facilities of TYPE are available within BACK so that any message may be given to the student before he is expected to make another response.

(v) CTRL. To facilitate the writing of lessons using Learner Control techniques suggested by Grubb (1968), the CTRL statement was created. In it, the author specifies four valid labels of any type whose addresses in the intermediate code are then loaded into four control address registers. Whenever the keyboard is unlocked to a student, entry of one of four possible pre-determined responses will cause the student to be branched to an address given by the corresponding control address register. The choice of the four pre-determined responses was "?F", "?B", "?S" and "?G" which indicate a forward skip, a backward skip, a skip to the subject outline and a skip to the glossary, respectively. However, these meanings are completely arbitrary and the author may decide on a quite different interpretation of the pre-determined responses. The only constraint placed on the author is that, for the statement to be effective, it must appear at least once in a segment. This is because the four addresses are retained in the registers until they are either replaced by another CTRL statement or lost upon entry into a different segment.

If the author has not specified a CTRL statement within a segment, then entry of one of the pre-determined responses results in either a restart in the case of a pause or an incorrect response in the case of a question.

2.6.2.8 Subroutine facility. To cope with the fact that certain parts of a lesson might be executed any number of times, a subroutine facility was considered necessary. The statements designed, however, do not allow the passing of parameters, for this was not deemed necessary, but did allow a return to wherever the author desired rather than to the point where the call was made. The explanation is as follows. Two different statements constitute the call. Firstly, there is a LOAD statement which specifies the return address, any valid label. Then, a JUMP statement transfers control. After the common statements have been executed, RETN transfers control to the address given in the last LOAD statement. In fact, there are five stacks of return addresses. Correspondingly, there are five different LOADn statements, each of which load the given address on the top of a push-down stack, and five different RETNn statements, each of which unload the address on the top of the corresponding stack and transfer control to it. In this way the author is able to employ numerous implied branches, including nesting up to 10 calls in each stack.

2.6.2.9 Other statements. For completeness, the STOP statement was added to terminate execution of a course. The only constraint placed on its use is that either itself or a TRANS statement must appear at least once in a lesson as either of them indicate the dynamic end of a lesson.

The END statement is non-executable and defines the static end of a lesson for the translater.

2.6.3 Use of the author language within NUTS.

There are three modes of operation of the author language translater that the author may request. He obtains these by his choice of operands in the BUILD command.

2.6.3.1 Mode 1. The author may wish to create a new lesson and enter his source statements after prompting of the line number. He may optionally give the starting line number for the statements in the lesson and the increment with which the line number will increase on successive prompts. After the system prompts, the author enters his source statement which is immediately parsed and intermediate code generated. In batch, the statement is listed on the author's print-out for possible debugging purposes.

If the statement is successful, the line number is incremented and the system prompts once more. If unsuccessful and from a terminal, an appropriate diagnostic error message is returned, together with a prompt to make a modification. At this point, the author may modify any previous line, delete any previous line, or insert any line in his lesson. If unsuccessful and in batch, the incremented line number prompt continues as if it had been correct after an error message is given.

The format for modifications is simply the line number followed by a comma and then the line contents. For deletion, the line contents would be null. One consideration when making modifications in this way is as follows : if the author corrects a line immediately he is told it is incorrect, then the processor may continue with one and only one pass, no re-translation of the whole lesson being necessary; otherwise, in the event of the modification of a different line from that given in the current incremented line number prompt, this line is checked syntactically within itself, but without reference to the rest of the lesson. The whole lesson is re-translated at a later stage. When the author has completed his modifications, he simply presses RETURN to go back to the incremented line number prompt.

On entry of the statement END, the translater completes such processing as segment address recording, label chaining, etc. Then, the author using a terminal is given the option for a complete or part source listing, whether he wishes to make further modifications, and whether he wishes to continue processing. In batch, none of these operations is available. If the translation has been successful, the command terminates. However, if

unsuccessful, then the author on a terminal has the further option for modifications, but in batch termination occurs with an appropriate message.

2.6.3.2 Mode 2. The author may wish to retranslate a previously existing lesson but first enter modifications. Modifications are then entered in the same format as described above. If an incorrect source statement is entered for a modification, then, from a terminal it may itself be modified, but, in batch, modifications cease at that point. Of course, this will produce invalid commands in the command sequence, all of which will subsequently be ignored. When modifications have all been entered, execution continues as in mode 1 with options for source listing, etc.

2.6.3.3 Mode 3. The author may wish to re-translate only a previously existing lesson. If the translation is unsuccessful, then the terminal author has the chance to enter modifications. In batch, termination occurs with the appropriate message.

2.6.4 The translater.

There are two distinct stages in the use of the author language : the translation stage and the control stage. Such a method was chosen instead of an interpreter because it was thought that student session time would be much greater than author preparation time. Thus, re-translation of source code every time it is encountered would be less efficient.

The translater produces an intermediate code which is stored away until the controller uses it to produce the instructional material. The reasons for producing intermediate code are as follows.

(i) Having produced a simple intermediate code it is relatively easy to devise a controller to execute it.

(ii) The intermediate code produced is independent of the machine being used. Thus, courses produced on one machine could be run on another.

(iii) The intermediate code is independent of the language that produced it. Therefore, future developments may allow a different author language to produce the same intermediate code.

(iv) It is much easier and therefore quicker than generating machine code. This was considered most important as it was essential to have the system operational as quickly as possible.

Every source statement produces at least one element of intermediate code. This element may consist of the code value only, as in STOP, or perhaps, a code value together with numerous other integer values, as in a TRANS statement where the lesson number and the segment number are also stored in the intermediate code.

A brief description of the translater logic will now be given. Mode 1 usage is assumed, for it is the most general.

A prompt together with a line number is given. Upon entry of a source line followed by RETURN, this is first stored in the appropriate place in the lesson file and a version without superfluous blanks is retained in a buffer for processing. Intermediate code is produced to identify the line number. This was adopted in order to give a line number with control time failure messages.

The buffer is then searched for a label whose identification is made easier by the fact that they are all terminated by a right parenthesis. After a label, a statement must follow. Some statements such as the response assignment statement and the array arithmetic assignment statement commence with special characters which makes them easily distinguishable. In the absence of a special character, the next characters in the buffer must be a string of letters for we have already precluded digits by having searched for labels. This string is then converted to a unique numerical value in the usual way. A test to see if the character immediately following this string is an "=" or not determines whether the statement is a non-subscripted variable arithmetic assignment statement or a code word such as TYPE, IF, etc. In either case, further extensive processing and appropriate generation of intermediate code occurs.

If the source statement is incorrect, a diagnostic error message is produced and the line number is placed in the error table. Then, a prompt occurs for a modification. This is read into a buffer and the line number and separator trimmed off. If the line is blank, control returns to the line number prompt. If the line number is that of the source line just entered and no out-of-sequence entry of source lines has taken place, then the pointer to the next available position in the intermediate code buffer is returned to its position before the entry of the erroneous line. Also, other information from the last line, such as label table entries, is discarded in order to recover from the error. The actual line is then stored and a version without superfluous blanks is processed as described above. As many modifications as is required may be made after which the line number prompt returns.

For a successfully translated statement, the error table is searched to see if the new version of the line can nullify such an error. The intermediate code is stored sequentially as it is produced in a buffer of 200 elements. A check is made at this point to see if, as a result of the last statement, the buffer is over half full. If it is, then the first half of the buffer is written to the lesson file on disk and the latter half is transferred to the first half. In this manner, only a fraction of the total intermediate code produced is ever stored within the translater and that contained on disk is stored in blocks of 100 elements.

Upon entry of END, the remainder of the intermediate code is written to disk. Also stored are the numbers of real and integer variables used and the addresses in the intermediate code of the start of each segment. Then, label chaining is carried out to assign to each incomplete label reference label addresses of those labels which were undefined when first encountered.

2.6.5 The controller. Whenever a user calls for a course, the controller takes the stored intermediate code from the appropriate lesson and executes the instructional material. The logic of the controller is described below.

It is first determined whether that part of the response file corresponding to the lesson at hand exists. This information is obtained from an index in the response file itself. If necessary, the appropriate initialisation takes place and the index is updated.

The restart segment is read from the response file and the corresponding address in the intermediate code picked up from the segment directory of the lesson. Two appropriate blocks of intermediate code are read in from the lesson file on disk together with the current values of all variables and stacks from the response file.

Control then passes to a section to which it returns at the end of execution of every single code. There, a check is made to see if the pointer to the current position in the intermediate code is indicating the second of the two included blocks. If it is, and if all blocks have not yet been read in, the latter block replaces the first one and a new block is read in in sequence. The pointer to the intermediate code is adjusted accordingly. In any event, the next element of the intermediate code is read and a switch decides where execution will be transfered depending upon the code. For certain branching instructions, when addresses become out of range of the two included blocks, two new blocks are read in.

When a STOP code has been executed or "?END" has been entered by a user, control passes to the terminating sequence, in which the time of day is obtained, the session time calculated from that and a previous call at the start of the controller, and the session time recorded in the response file.

2.6.6 Implementation techniques.

2.6.6.1 Labels. To each type of label there corresponds a label table.

Each entry in such a table indicates whether that label is unused, or, if used, gives the address in the intermediate code of its occurrence. When reference is made to a label, the corresponding address is brought from the table, if it exists. If not, a chain value is entered in the intermediate code. This chain value contains a backward pointer to the previous position where an unresolved label reference is pending and the currently required label. The last reference in the chain has a zero pointer. After entry of END, when all labels should be present, chaining backwards through the intermediate code occurs, and the missing addresses are added.

2.6.6.2 Arithmetic expressions. The input string is converted to reverse Polish. Each occurence of an object, that is, constant, variable or standard function reference, generates several elements of intermediate code, whereas operators produce only the corresponding code element. The algorithm used for conversion to reverse Polish is as follows. If the priority of the current operator is greater than that of the operator on top of the stack then stack the current operator; otherwise unstack, placing the operator from the stack in the intermediate code, then test the priority of the current operator with that now on top of the stack. For a left parenthesis, which has the lowest priority, always stack it; for a right parenthesis, unstack all operators into the intermediate code up to the next left parenthesis, which may now be discarded.

Constants are analysed using a state table technique. So that the intermediate code produced is of mode integer, the code stored is of the form integer mantissa then exponent. When a variable name is encountered, a search of the hash table occurs. If it is the first occurence of that name and if there is no ambiguity between an array name and a non-subscripted variable name, then a store address for that variable is assigned. Any future reference to that name will use the same store address. Only the code for the particular standard function is generated. That for the parameter expression precedes it. Array subscript expressions are treated in the same way.

At control time, execution is performed using a work stack. Only the top two elements at most are changed by an operation code. For expressions, the codes consist solely of those for fetching variables and constants, calling the standard functions and executing the arithmetic operations. With standard function parameters and array subscript expressions, the value is at the top of the stack when it is needed.

2.6.6.3 Response assignments. For each response assignment, the translater produces many different intermediate operation codes. A code is generated for the type of response element followed by the actual element number. Then for every string and/or value specified, code is generated as follows.

A string is recognised by a single quote in the input buffer. Every character between the pair of single quotes is recorded in the code, preceded by the number of such characters. Appended to each string is a code and value to indicate with what error the string is to be matched. A value may appear on its own, in which case no error is to be tolerated in the numerical match, or be followed by a second value which is to be the error bound. The code produced for both of these is composed mainly of that already mentioned in connection with arithmetic expressions. The exceptions are that in the first case a value code and a default error term code follow the code generated for the value expression whereas in the second case, code for the error expression is followed by an error code.

At execution time, two buffers, one each for strings and values, are loaded with all the response assignments before the particular response acceptance request. Of course, at this stage, the actual values for values and error bounds are known and it is these that are loaded into the value buffer.

2.6.6.4 TYPE/BACK. The translater recognises strings and arithmetic expressions during a TYPE statement in much the same way as for a response assignment. The notable addition is in the format specification for these two types. At the end of the statement, the format information is converted to form a FORTRAN dynamic format using the FORTRAN T format to

specify starting positions, the FORTRAN literal format for the strings and FORTRAN I, F or E formats as appropriate for the expressions. The dynamic format is packed four characters to a word and then stored in the intermediate code preceded by the number of words.

At execution time, the dynamic format is read into an array. Taking into account the constraint that a maximum of five expressions may be output in any TYPE statement, the number of different combinations of real and integer list elements for 0 to 5 expressions is 63. Thus, the controller contains 63 FORTRAN output statements, one of which is appropriate for each combination of list elements. This method may seem a little crude and, indeed, has the limitation that only a fixed number of items may be output, but implementation is so much easier than it might have been if non-FORTRAN coding had been used, as it certainly would have to have been for a more sophisticated output statement.

2.6.6.5 RESP. In its simplest form, the statement consists only of the keyword and in that case only an intermediate operation code is produced. However, variables may be specified to contain the numerical values that the response might contain after entry. These are dealt with in three different ways. For a non-subscripted variable or an array element, the hash table is used to obtain a store address for the name. An array name followed by a starting position is the last possibility. Here, store addresses are needed for both the array name and the starting position, unless, of course, this is given by a constant.

Despite the fact that the translater has an easy task to analyse a RESP statement, the controller is faced with a considerable amount of work at run time. The keyboard is unlocked to allow a response to be input. The last non-blank character is located and, if it is the continuation character, then another line is read in. Up to four such continuation lines are allowed and the whole response then resides in the response buffer. The beginning of this buffer is then searched for the four pre-determined responses connected with the control address registers. If one is present and if a

CTRL statement is in effect, then transfer of control occurs. Otherwise, the buffer is searched further for other pre-determined requests. If "?END" is requested, then the session ends and control returns to the command language interpreter. The other two possibilities are "?CALC" or "?PROG". In each case, the user is prompted for appropriate information regarding which program and what mode of use is required. Upon return to the lesson, the user is prompted to make the response that he used instead to request the calculating language.

If no request or pre-determined response was entered, response processing continues as follows. The string buffer contains all those strings indicated during the response assignment statements, the degree to which they have to match the response, the number needed to give an overall match and whether ordering is important. For each separate response assignment statement, the strings are compared with the contents of the response buffer. In this way each of the response elements can be found to be true or false with respect to string matching. In order to save time, there is a forward pointer in each part of the string buffer so that comparison will cease when the required number is obtained. Alternating with the string comparison for each response element is that for values using the value buffer, whose organisation is much the same as for the string buffer. However, when the first value comparison has been indicated, all values contained in the response are immediately processed and stored sequentially in an array. In this way, the values are converted from input characters only once and subsequent comparison involves only a search through this array. Thus, each of the response elements is found to be true or false with respect to value matching. If both types of matching are true then the response element itself becomes true.

If variables were specified for the storage of any numerical values which might be contained in the response then the values are available in the correct sequence in the array. These are transferred to the variables,

a large positive constant being substituted for a shortage of values.

Successive calls for time of day before and after the response was made give the response time. This and the truth values for the response elements are then stored in the appropriate part of the response file. If all the response elements are false, the first 80 characters of the response are also stored for future reference. This is an unanticipated response.

2.6.6.6 Logical expressions. As for arithmetic expressions, the input string is converted to reverse Polish. Here an object may be a response element, either implicit or explicit, a reference to the past student performance facility, or two arithmetic expressions separated by a relational operator. The algorithm used is the same as before.

There are eight different kinds of enquiry that may be made during a #PERF request. It is also possible to test the truth of the conjunction of any number of these eight types. Thus, the translater produces nine "subcodes" for this facility, the extra one denoting the end of the request. One extra piece of information stored is the address of the last element of the intermediate code produced for the call. This is most useful as time can be saved if any one enquiry proves false, therefore making the whole request false, and rendering the rest of the enquiries useless. Execution passes straight to the next operation code in this instance.

. At control time, the same work stack is used as for all the arithmetic operations. The only difference is that true is represented by the value 1 and false by -1. To provide protection in the use of logical quantities, a value "undetermined" may be returned to the stack. This occurs if a particular response element has not been used yet referred to or if the #PERF facility is being interrogated about a question for which there is no information. In this case a run time failure occurs and the appropriate diagnostic error message and line number is returned to the author for debugging purposes.

2.6.6.7 IF. After the code generated for the logical expression, there is the IF operation code followed by a blank element, which can only be filled in after the translater has processed the executable part of the IF statement which may be any other valid statement except another IF.

At execution time, the executable part is obeyed depending upon the truth of the logical expression. This value is on top of the stack. If it is false, the instruction address pointer takes the address of the next statement, as mentioned above.

2.6.6.8 TRANS. It is a simple matter for the translater to generate the operation code for this statement followed by the lesson number and the segment number but not so straightforward at run time. If the user is a student, then a table lookup in the lesson index occurs to see if the new lesson is released. For an author, a table lookup on his file catalogue indicates whether the new lesson exists. In either case, an MTS restart command is generated so that after NUTS execution pauses and restarts the new lesson file may be accessed. The segment index then shows whether the segment specified does exist. If it does, the corresponding address is obtained and the current position information in the response file updated.

2.7 The calculating language.

2.7.1 Design considerations.

In a CAI environment, it is essential that the student be able to call upon calculation aids during a course of instruction. Certain questions he may be asked could require more computation than can reasonably be expected by hand calculation. In an effort to surmount this problem, NUTS supplies a simple computing language, the programs of which may be stored for subsequent use. The language may be used in command mode but its primary importance is use during a course whilst answering a question. The student may write his program on either occasion and may call upon it, even making alterations as necessary, within a course.

The design considerations of the calculating language in many ways resemble closely those for the author language. Consequently, in this section, only those features which differ from those of the author language are pointed out.

2.7.1.1 Labels. Statement labels are provided in the calculating language to reference one statement from another by branching. They may be assigned in any order without affecting the order of execution of a program. However, neither question labels nor segment labels are meaningful in this context and are not available.

2.7.1.2 Output. The identical capabilities outlined in 2.6.2.3 are available using the TYPE statement. However, no PAUSE exists as that facility is meaningless in the context of mathematical computations.

2.7.1.3 Input. The READ statement was included so that the user may enter any data he requires. He specifies the variable into which the data entered at run time is placed. Three methods of specifying the variable are allowed. Two of them quite simply are a non-subscripted variable or an array element. However, a third was devised in case a large number of numbers was to be read in. This entails specification of an array together with a starting position, which itself can be either

a constant or a non-subscripted variable. Thus, any data read in, which may be a variable amount, is stored in successive array elements starting at the position indicated.

2.7.1.4 Conditional expressions. To allow conditional branching and, in particular, the building of loops in a program, conditional expressions are included. Logical quantities, available for such expressions, are formed from two arithmetic expressions separated by a relational operator. Logical operators are available to form expressions from these quantities.

2.7.1.5 Branching. The branching facility comprises IF and JUMP. TRANS, BACK and CTRL are author language statements only.

2.7.1.6 Other statements. Other statements include STOP, which represents a dynamic end to a program, and END, which is non-executable and only serves the purpose of informing the translater that the static end of the program has been reached. No subroutine facility was included as the programs generated were expected to be so small as not to warrant such an addition.

2.7.2 Use of the calculating language within NUTS.

Substituting the command CALC for BUILD in 2.6.3, the description of the use of the calculating language within NUTS is otherwise identical to that for the use of the author language. The only additional feature is that after a successful translation of a program, it is immediately executed. For further execution, the PROG command is used.

2.7.3 The translater and controller.

Like the author language, there are two distinct stages in the use of the calculating language: the translation stage and the control stage. The logic of the calculating language translater and controller is similar to that of those of the author language.

2.7.4 Implementation techniques.

The techniques used to implement the calculating language correspond exactly to the techniques for those statements which are the same in the author language. The only additional statement not covered in 2.6.6 is

READ. Three different types of variable specification may occur within a READ statement. When the translater recognises one of these therefore it generates the operation code for READ followed by the "subcode" for the particular specification. In the case of a non-subscripted variable, the hash table is searched and the store address written into the intermediate code. In order to provide a prompt at run time, the variable name, preceded by its length, is also stored. For an array element, the code is first generated for the subscript expression in the usual way. Then, the store address and the characters in the array name are determined and stored. The third type involves the specification of a starting position along with the array name. This starting position is only optional, however, and defaults to the first element of the array if absent. In either case, the code generated in the first instance is that for the store address, the length of the array name and the characters in the array name. The code which determines the starting position is then generated.

At run time, when a simple variable is to be read in, the user is prompted with the name of the variable. Rounding occurs, if necessary, after a warning message. When an array element is needed, the user is prompted for the actual element number required, that is, the system has already evaluated the subscript expression. When a sequence of array elements is to be read in, the system prompts with the array name followed by the actual starting element number. The user enters as many constants as he desires and these are stored in consecutive elements of the array. He separates them by commas but terminates the sequence with a semi-colon. As the constraint that only 80 characters may be entered in one line is imposed on the user, he may continue on another line by neglecting to terminate the previous line correctly. The system will then prompt for the next element in sequence. If an error is detected in a line of constants, the user is asked to re-input the current line. Entry of commas only in sequence causes zeros to be entered in the

corresponding array elements. Where rounding occurs, the user is warned for which element this has been carried out.

2.8 The desk machine.

To enable a NUTS user during a lesson or, more important, a student about to answer a question during a course to make a simple "once-only" calculation, it was decided to include a sequence controlled desk calculator.

2.8.1 Design considerations.

To allow calculation and a very limited storage facility, there is one accumulator and 100 storage registers. All numbers, whether written as reals or integers are held in floating point form. The following instructions provide the ability to perform simple calculations as desired. The operand X denotes either the number of a register between 1 and 100 or that of the accumulator which is 0. Where meaningful, this operand may be replaced by an actual number.

(i)	I	X	#number	-	initialises register X to the number.
(ii)	T	X		-	types out the current value of register X.
(iii)	L	X		-	loads register X into the accumulator.
(iv)	U	X		-	unloads the accumulator into register X.
(v)	A	X		-	adds register X into the accumulator.
(vi)	S	X		-	subtracts register X from the accumulator.
(vii)	M	X		-	multiplies the accumulator by register X.
(viii)	D	X		-	divides the accumulator by register X.
(ix)	E	X		-	raises the accumulator to the power given in register X.
(x)	H			-	terminates execution of the desk machine.

2.8.2 Implementation.

Instructions entered for the desk machine are executed immediately. The whole line is read into an input buffer which is then stripped of all superfluous blanks. What remains in the buffer is easily recognised by the fact that the instruction code is a single letter, a storage register including the accumulator is an unsigned integer of one to three digits

and a number, signed or unsigned, is preceded by a "#".

Once these parts have been recognised, a switch directs control to the code for the particular instruction and the values of the accumulator or storage register which reside in an array are updated accordingly. Diagnostic error messages are returned for misconstructed instructions and for division by zero and any exponentiation operands that would give an infinite, imaginary or indeterminate result.

2.9 The Pittsburgh Interpretive Language, PIL.

2.9.1 Reasons for its inclusion in NUTS.

To use either of the two calculation facilities provided, especially the calculating language, the user has to learn the elements of the language and then it may take him a short while to become proficient in using it. It was therefore decided that it would be a great advantage to include in NUTS a simple language, easy to learn, but with those powerful features that were missing from the calculating language, such as string manipulation, extended input/output, etc. If such a language existed already in MTS, then the fact that a NUTS user may already have some knowledge of it would also recommend its inclusion. The language that fitted these requirements was PIL. It is also suitable for use in a learning environment as it possesses error recovery capability, which is most suitable for use in testing algorithms and other allied problems.

It was also essential to allow users the facility to store PIL programs from one call of the interpreter to the next. Therefore, one file is set aside for this purpose. Naturally, if a number of different programs are to be stored then the user is burdened with remembering which PIL parts they are in, for these must, of necessity, be unique.

PIL possessed a most powerful "direct" mode, which is equivalent to a desk machine. Consequently, the inclusion of PIL tends to make the sequence controlled calculator, and, to some extent, the calculating language, somewhat superfluous, although, for ease of storage of programs, the calculating language is still of use.

2.9.2 The programming language.

PIL was designed and implemented at the University of Pittsburgh. It is a remote terminal language, designed to make advantage of the computer-user interaction made possible by terminal interface in a time-sharing or multiprogramming environment. Although similar to earlier conversational languages such as JOSS and BASIC it has major differences in debugging

facilities, error reporting and problem solving capabilities.

PIL provides the user with much greater assistance than the usual batch compilers offer. This includes the use of terminal diagnostics, user interaction with the machine and associated error recovery procedures. A major goal of the language was that errors be recoverable. To a user, this means that it is possible to sit down at a terminal with a problem and work towards a solution. As he becomes aware of the need to make corrections or improvements, PIL allows him to alter his program and to continue without requiring a new start (the PIL "GO" statement). PIL sacrifices machine efficiency in the hope of gaining increased human efficiency. It is a clear, unambiguous language, quite easy to learn, so that students and researchers alike quickly master the language and its use.

For simple computing tasks, PIL typically generates answers as fast as the user at a terminal can assimilate them. As more complex tasks are programmed, a point may be reached where the user regards the performance of PIL as less than ideal. The response time may become too slow or too much computer time may be used.

Unlike the usual compiler languages such as FORTRAN, ALGOL or PL/I, PIL does not translate a symbolic program into a machine language program for later execution. In PIL, the original symbolic statements are maintained in storage and these statements are interpreted each time they are executed in a PIL run. Thus, it should be recognised that PIL is not the appropriate vehicle for large "production" jobs. It may be of value, however, to code such a program initially in PIL. The debugging facilities can be used to check out the logic of the algorithms for such jobs. Then the code should be transcribed into a compiler language for more rapid execution.

A full description of PIL is given by Flanigan (1968).

2.9.3 The implementation of PIL within NUTS.

The PIL interpreter used is simply a copy of that available in MTS

as a library file. The only additions made are to provide an interface with NUTS. A brief description of this appears in 2.5.5.10.

2.10 Performance.

2.10.1 General statistics.

NUTS comprises a main program and 40 subroutines coded in FORTRAN IV, and three subroutines written in assembler. The total number of source statements is around 9000. The assembler routines are system dependent and are used to return the clock time, indicate whether the user is performing a conversational or non-conversational job, and destroy a named file. In use, the whole system occupies 80 virtual pages on the drum.

2.10.2 Current use.

After initial test and demonstration programs for checkout purposes, NUTS was first used in an investigation into the production of instructional programs for elementary electricity and magnetism (Adams, 1969). The main application of the system to date is the PIL programming course, a full description of which appears in Chapter 3. As well as being used for the course, the instructional programs are available for general use to anyone desirous of learning PIL.

2.10.3 Possible future developments and improvements.

It has been mentioned earlier that NUTS was designed and implemented in as simple a way as possible so that an operational system on which to carry out further research could be available as soon after the commencement of the project as possible. For this reason, reflections on the design and implementation have suggested that the system can be made to run more efficiently with some alterations. Also, some additional features would be most useful. Some possible amendments are given below.

With respect to the implementation of NUTS, increased efficiency in the future demands that shared code be used. At present, the virtual memory integral (VMI) in the wait state is enormous, which gives quite an overhead. To alleviate this problem by some other means, if that were possible, would necessitate a study into the use of dynamically loaded routines. In other words, when NUTS is initiated, instead of all the routines being loaded in

at that point, only the command processor would be loaded and, depending upon which command was requested, other routines would be loaded in when required. When these routines had been used, the command processor would be reloaded. Certainly, this organisation would lend itself to much smaller virtual memory requirements and hence less paging, but whether this would offset the increased use of the loader is a matter for conjecture. Unfortunately, at this point in time, the data collection by the general operating system does not provide information for such a study.

One additional facility that would enhance the author language would be the introduction of string variables and functions. This would give the ability to manipulate the response, which would be automatically stored in a predetermined variable, using string functions such as editing characters, concaternation with other strings etc. This would allow the author to reply to the student in the actual words of the student's responses. The main reason why this facility was not implemented was that such complex string manipulation as was visualised is difficult to execute using FORTRAN. Also, the extent to which this particular part of the language would be used was not considered great enough to warrant the time it would take to include it.

Increased efficiency in the author language processors would be gained if the literals used during response assignments and output statements were stored as characters and not as their equivalent numerical representation in the intermediate code. Also, the response buffer used during response analysis would be better created when the response assignments are parsed, not in two stages, both during translation and interpretation of the response assignment statements.

A more detailed appraisal of the system, in the light of the investigation which included the PIL course, is given in section 3.4.

CHAPTER 3 An investigation into the use of NUTS to teach a programming language.

3.1 Previous attempts at programming courses.

3.1.1 Other course structures.

Very few programming language courses have been given using CAI. This is surprising in that there is an ever-increasing number of academic disciplines pervaded by computer-based techniques and hence the overall demand for people who can program is growing very rapidly.

In order to become reasonably proficient, considerable practice is needed and so adequate opportunity to run programs must be given. Of similar importance is the necessity for this practical experience to be coordinated with the theory behind the students' problems. This requires supervision, which, under normal teaching circumstances, puts a heavy demand on lecturing staff whose time is considered at a premium. The investigation that was chosen was a comparison study between various methods of teaching a programming language. It was hoped to ascertain whether it is feasible to use CAI to teach a programming language and hence free academic staff for more research.

Some previous attempts at such courses have used a computer in the learning process but not for CAI as defined in this thesis. SCOCP, Student COntrolled On-line Programming (Lambert, 1968), assumed that a student has learnt the rules of a programming language and that what he needs is practical experience in manipulating these rules. On a teletypewriter, the computer provides the student with the content of storage locations upon which his program is going to work. As the student types in his list of coded instructions, the computer checks that they are acceptable and carries them out at the student's command. No correct answers are stored, but the resulting values of the storage locations are returned for the student's perusal. Conversationally, the student is also able to edit his program, step through the execution one instruction at a time and display various locations. A request for help causes a demonstrator to come to his aid.

A similar project at Brighton College of Technology (ICL, 1969), but non-conversational, provides FORTRAN tuition by private work on a programmed text. Then, using standard software but given pre-punched cards for easy entry, the student compiles and tests example programs. When satisfied that a program is working correctly, he submits it to the computer for judging by comparison with results for standard data. Facilities for teachers to put different programming problems into the system and to obtain information on project progress by student or by problem are provided.

In other projects, programming languages have been specially designed so that actual programming techniques can be taught, not simply an existing language.

Lorton and Slimick (1969) list the advantages of teaching a "symbol manipulation-list processing" language. However, as some machine level concepts might be usefully included in the course, they provide also a simple assembly language. To this end they designed a driver program to supervise the interaction of the student with the curriculum material and the language processors, an interactive assembly language processor and an interpretive processor for the "symbol manipulation-list processing" language. As well as providing communication with the language processors, the driver program presents the instructional text and evaluates responses. The symbol manipulation-list processing language used is a dialect of LOGO (Feurzeig and Papert, 1968).

Whilst designing their TEACH system, Fenichel et al. (1970) considered that their chief objective was to teach programming, not a particular language, but realised that some language was required as a vehicle to convey the ideas. In fulfilling their design criteria, they devised an interpretive language resembling JOSS that

(i) allowed students to adjust easily to any standard algebraic language afterwards;

(ii) contained all the fundamental ideas of current programming practice; and

(iii) allowed presentation of an important idea only after a need for it had been established.

They surrounded the language processor with a teaching system which presents lessons to the student, supervises his progress and permits him to exercise his skills. The teaching system language generates "scripts" and is quite general, allowing arbitrary recursion, conditional transfers, scanning of student input for keywords, etc. The "scripts" have little control over what the student does with the programming language interpreter but an important feature is that the syntax scanner of the interpreter will not recognise any construction which the "scripts" have not already discussed. Otherwise the scripts engage in a limited dialogue with the student, allowing him to request hints about the suggested problems and to determine whether certain sections should be skipped.

At the University of Texas (Homeyer, 1970), using the PLATO instructional system under the RESPOND time sharing system on a CDC 6600, a CAI course has been developed to teach assembly language programming. A language called ELASTIC was designed. It is written in FORTRAN and is composed of an assembler and an interpreter, self-contained and can be executed, with minor modifications, on any machine with a FORTRAN compiler. The capabilities of the ELASTIC assembly language may be presented in segments of gradually increasing difficulty and sophistication and hence the ELASTIC system is divided into four "computers" to facilitate this pedagogical approach. The main feature of the course is that execution of programs written in ELASTIC is allowed at any time during a CAI teletype session.

There have been a number of courses, however, which have taught existing programming languages.

Schurdak (1967) attempted to measure the effectiveness of CAI in teaching the FORTRAN language. He used 48 students and divided them equally into three groups, one group taking conventional instruction, another taking a programmed instruction text only and the third taking CAI. The students were further classified in other subgroups such as paid/not paid, graduate/undergraduate, etc. Using an aptitude test and a one-day retention test, and with a CAI program that was basically linear drill-and-practice with some immediate effective correction procedures, Schurdak reported a significant difference, at the 1% level, of criterion test performance for the CAI group, as well as a similar standard deviation of test scores for this group. Students with high pre-test scores appeared to score approximately the same on the post-test, but, at the other end of the aptitude test score scale, it apparently made a very large difference as to how a student was instructed in FORTRAN, even within a superior sample such as university students. Two important points should be mentioned about this course. Firstly, the CAI group receives their initial presentation of material from a well-known text on FORTRAN. Their understanding of the material is tested on a terminal. If the student meets the test criteria, the computer indicates that he should read the next lesson but if he is uncertain of the subject matter, but still passes, he has the option of receiving further questions but not explanatory text. If the student does not meet the test criteria, he receives a series of diagnostic drill-and-practice questions, which have all to be answered correctly before he may proceed. Secondly, if the text material is not adequate, the computer gives the student additional explanatory or remedial material taken from a well known PI manual. Otherwise, the computer does not give any textual material.

Gross et al. (1969) carried out a similar investigation into the teaching of FORTRAN using APL as the instructional language. 76 students were divided into a CAI group, a PI group and a group who received

conventional lectures and classes. Distribution into groups was carried out with respect to student choice as far as possible, the only limiting factor being that the sex ratio was consistent in each group. An analysis of variance on the aptitude scores revealed that there was no significant difference between the groups initially. During the course, the learning curves of the first group of CAI students are used to develop a non-linear sequencing of course material. A heuristic learning model uses cumulative frequency distributions of all student responses to different combinations of questions to decide the sequence of questions for any one student. Performance scores for each group were obtained from three FORTRAN tests and four FORTRAN problems. Only on the first two tests did significant differences appear. However, an examination of mean scores indicated it was the PI group that gave a superior score. The source of the superiority is suggested by the total time data. The average course completion times for the CAI and conventional groups were about one third and one half respectively of that for the PI group.

Other courses developed to teach programming languages include:

(i) The University of Texas at Austin use a course to teach COURSEWRITER I using tutorial logic. It was written at Florida State University in COURSEWRITER I.

(ii) Pennsylvania State University have a course which presents information regarding the various instructions in COURSEWRITER II. It is written in COURSEWRITER II and has an adaptive strategy.

(iii) For prospective authors of courses in CAILAN, the CAI laboratory at Harvard have a course on CAILAN written in CAILAN and using both Socratic and tutorial dialogues.

(iv) A course to teach COBOL is under continuing development at the Human Resources Research Office (Hum RRO) in Alexandria, Virginia.

(v) The Aerospace Corporation of El Segundo, California, are developing a FORTRAN course under the PROCTOR system which monitors the interaction between the student at the terminal and the instructional program on disk. Tutorial logic is used and students are permitted to enter FORTRAN statements for checking by the FORTRAN compiler, receiving diagnostic error messages as appropriate.

3.1.2 Significant features of this course.

It was decided that an existing programming language should be taught in the course. This would allow for future use of the language by the students as it is available under Michigan Terminal System. Also, the fact that the language processor was available for modification, if necessary, was taken into account.

Two comparisons on the relative effectiveness of different teaching methods were desired. Firstly, CAI was to be compared with conventional teaching. Secondly, for examples sessions, on-line practical classes were to be compared with the conventional demonstration classes. As a result, three groups were needed (see 3.2.1).

The CAI part of the course was divided into two parts. The lessons, which presented the instructional material to the student and continually questioned him as to his understanding of it, were structured using an adaptive tutorial logic. However, the examples sessions, which were interspersed with the lessons, contained a linear sequence of problems but used a learner control strategy within problems. This latter choice was necessary in the light of the second comparison as mentioned above. Normally, during problems classes, control is with the student. He decides what resources he needs to tackle the problem, when he needs help, and when he wishes to test his solution. Consequently, two most important facilities were placed at the student's disposal during the problems classes.

These were the language processor which he could request at any time and which stores all his previous statements for that particular problem and a comprehensive help facility which returns the steps into which the problem may be divided and either programming or logic help with any of the steps. All the student has to do in either case is inform the machine exactly what he requires.

3.2 Description of the course.

3.2.1 Background to the course and the students on it.

Irrespective of whether they intend to do Honours Maths or Honours Computing Science eventually, students in their first year are required to take a qualifying course in Mathematics as their main subject. (The curriculum is being revised for year 1970-71 so that there will be a first year course in Computing Science for potential Honours students.) However, during the last week of the second term, an introductory course to Computing Science is held, the main purposes of which are that the students find out a little about what their future course holds in store for them and that the staff of the Computing Laboratory find out a little about their potential in computing. In addition to those students who entered University to read Computing Science and are therefore required to attend the course, an invitation is given to potential Honours Maths students who may wish to switch to Computing Science.

The course lasts for five afternoons; in fact, when the students are available. In the three previous years when the course has been given, the format has been simply:

(i) the first afternoon comprised a lecture on "Introduction to Computing" followed by a programming aptitude test;

(ii) during the next three afternoons a programming language was taught; and

(iii) on the last afternoon a talk on "The Computing Science Honours Course" was given, followed by a display of the equipment.

For the first two years, the language taught was ALGOL, the teaching language of the laboratory. A punching service was provided and every student was expected to have made at least one program run on KDF9 during the week.

With the advent of Michigan Terminal System on the IBM 360/67, it was thought that the students would benefit more by seeing their programs entered from a terminal and the appropriate results and/or diagnostic error messages returned. A demonstrator sat at a terminal and typed in the students' programs. Upon detection of errors, the student indicated his intention but when in difficulty the demonstrator was at hand to give help. The language needed had to be a simple, easily-defined, terminal language with good diagnostic error messages. That language was PIL.

It was intended to ascertain the validity of NUTS for use in CAI and to discover whether the use of CAI together with an on-line processor was feasible in the teaching of a programming language. Thus, the five afternoon course for potential Computing Science students seemed to fulfil these requirements. Other reasons that recommended its use were:

(i) it was important that there would be a sufficient number of students available to provide the required number of groups yet at the same time that the groups would be small enough to allow the use of the small number of terminals available;

(ii) NUTS could easily be modified slightly so that PIL could be called from the author language, instead of its being available only by virtue of a NUTS command or a pre-determined response;

(iii) the course was long enough to be able to make some broad conclusions and recommendations for future use yet short enough not to disrupt the use of the available terminals in the laboratory; and

(iv) although the course usually provides useful extra information about potential Honours students, any untoward events during the course would not disrupt the selection process as the examination score at the end of the year is the vital factor.

The students were divided into three groups:

(i) group A, those who would have conventional lectures followed by the usual examples class - the control group;

(ii) group B, those who would be given lessons of CAI material followed by the same examples classes as group A; and

(iii) group C, those who would receive the same instructional material on the terminals as group B but would then have CAI examples classes in which the PIL interpreter would be available to them on-line.

The list of enrolments for the course finally totalled 22, with 12 being potential Computing Science honours students, and the remaining 10 potential Maths honours students who indicated their interest in the Computing Science course. The fact that seven terminals were made available in the laboratory for the week of the course and that almost all the students were available until 1800 each day, determined the following timetable.

Group A: lecture 1400-1515, examples class 1545-1700.

Group B: CAI lesson 1400-1530, examples class 1545-1700.

Group C: CAI lesson followed by CAI examples class 1530-1800.

Group A contained eight students and groups B and C seven each.

3.2.2 CAI content of the course.

The CAI course designed for three afternoons of the week consisted of two parts; that of three lessons which both group B and group C were given and that of three examples classes which only group C received.

The first difficulty experienced in coding the lessons was concerned particularly with lesson one. It is a well known fact that students must be taught quite an appreciable amount about a programming language before they can attempt the easiest of problems. To teach PIL, the topics which should be covered before an example may be attempted are:- a general description of PIL, direct mode, arithmetic, logicals, simple I/O, indirect

mode with parts and steps and running stored programs. Consequently, care has to be taken not to make this first lesson excessively long particularly since students in group B are expected to have been taught an equivalent amount as those in group A when they start the first examples class. The greatest difficulty was that the length of the first lesson could not be tested in the same conditions as the students would find, that is, seven terminals running the course whilst the machine was being used with its normal load.

Another point to consider is that, owing to time considerations, any questions asked the student have to be fairly simple and straightforward. However, at the same time, as the subject content is rather high, the instructional material in this first lesson must be liberally interspersed with pauses for reflections and questions for clarification.

Thus, for the most part, questions asked during that lesson are of either multiple choice or simple constructed response type.

The second and third lessons dwell upon particular aspects of the language and hence more advanced and therefore more interesting questions can be asked. The second lesson contains: the IF statement, implied loops, explicit loops and restarts. As a result, questions are asked of the student that require entry of a PIL statement. This is checked by NUTS and not the PIL interpreter. Another type of question included is that of following the course of a series of statements (usually containing a loop) and supplying values of variables at certain stages. The last lesson deals with character strings, string functions and extended console I/O. As in the previous lessons, multiple choice and simple constructed response questions are used but much more remedial branching is included.

During each lesson, the student receives a lot of feedback to his responses. To ensure that he is not misled by having guessed the correct answer each feedback message for a correct response usually reinforces the reason for the success. Except for a binary choice, in which case an incorrect

answer is commented upon in full, an incorrect or partially correct response usually receives a hint as to its invalidity or incompleteness and the student is invited to try again. At most, three attempts are allowed before a full explanation is given. In addition to feedback messages for each question, where there is a sequence of questions, the student is given his score over the sequence, together with an appropriate comment. Similarly, over lessons as a whole, comments are made on scores for such sequences.

Each of the three examples classes that group C are given contain four problems. These examples are given in sequence but once at work on a particular problem the student is given the freedom of learner control (Grubb, 1968) to choose the next facility he requires. Decisions such as when to use the PIL interpreter, what steps the problem may be divided into, what help is needed next, when to attempt an answer, etc. are all left to the student. The various paths that a student may take within a problem are best described by a flow diagram (Figure 3.1).

Each double-lined decision box represents a decision stage for the student. The decision stage of primary importance is that which the student encounters immediately after the statement of the problem and to which he returns frequently during the solution of the problem. There are four choices.

(i) INFO. This gives the student a numbered list of the sections into which the problem may be logically divided. Such sections as input of data, output of results, setting up a loop, etc. might be included in the list.

(ii) PIL. This provides the student with the PIL interpreter. If he wishes to ask for help or simply return to the primary decision stage, he enters "MTS" to the interpreter. All his source statements will be saved in a file related to that particular problem so that the next PIL request places them at his disposal without re-typing.

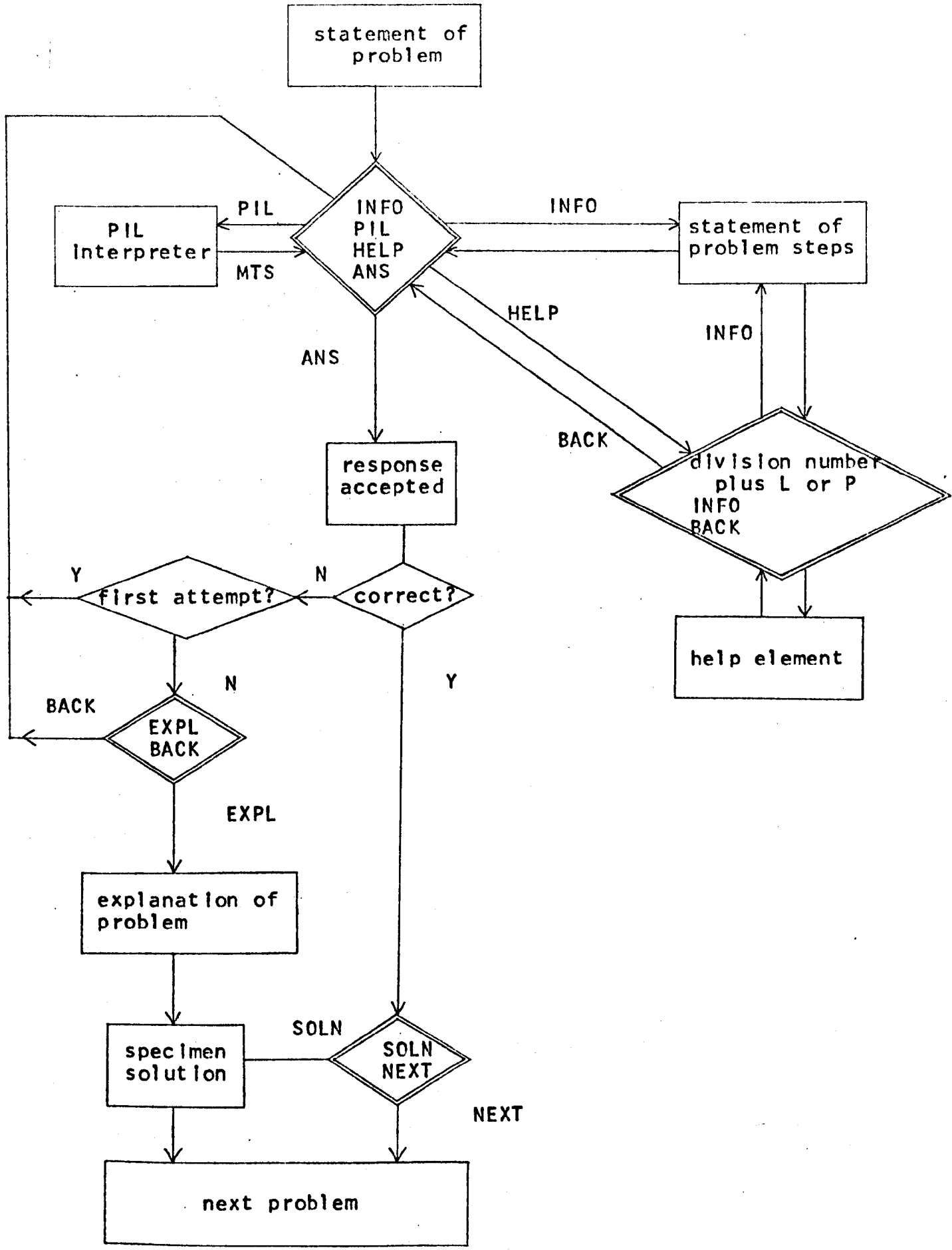

statement of problem
INFO
PIL
HELP
ANS
PIL
PIL Interpreter
MTS
INFO
statement of problem steps
HELP
INFO
ANS
BACK
division number plus L or P
INFO
BACK
response accepted
help element
Y
first attempt?
N
correct?
N
Y
BACK
EXPL
BACK
EXPL
explanation of problem
SOLN
specimen solution
SOLN
NEXT
NEXT
next problem

Figure 3.1

(iii) HELP. A secondary decision stage is reached where the student requests what type of help he requires.

(iv) ANS. The student is ready to submit his answer to the problem. A constraint that he must have accessed the PIL interpreter at least once is placed upon the student before his request to enter his response is accepted.

During the HELP decision stage, the possible requests are as follows.

(i) INFO. This gives the student exactly the same list as already described. It is included within the HELP stage as the numbers are required to obtain further help and it would be a waste of time for the student to have to return to the primary decision stage to obtain these.

(ii) A division number together with "L" or "P". The student may obtain further information about a particular section of the problem by stating the number given to that section by the INFO output. Two types of help may be obtained; help with either the logic or the programming of the section of the problem. Thus, either "L" or "P" must be given with the division number.

(iii) BACK. This merely returns the student to the primary decision stage.

The request to enter an answer is treated as follows. The response is accepted and tested for correctness. If it is correct, then the student may choose to receive a specimen solution of the problem before continuing to the next problem by entering SOLN or simply to proceed to the next problem immediately by entering NEXT. If it is incorrect, the course of action depends upon whether it is the first attempt at an answer or not; for a first attempt, the student is returned to the primary decision stage but for a subsequent attempt, the student chooses to return to the primary decision stage by entering BACK or to receive an explanation of the problem

followed by a specimen solution before proceeding to the next problem by entering EXPL.

To facilitate the running of the course by the students during the week, certain additions and slight modifications were made to NUTS.

Normally, the PIL interpreter is available to the NUTS user either through a NUTS command or through a predetermined response during a COURSE command. However, in either case, statements are only stored in the one PIL file which would become rather perplexing when 12 problems are to be attempted. The course of action chosen was to provide the PIL interpreter whenever the student requests it in the examples class and to supply a separate PIL file for every problem. This entailed introducing a further statement to the author language which simply called the PIL interpreter. Also, this statement would include the number of the problem currently being attempted.

As the student would not need any other facility but the COURSE command, a version of NUTS was used which only allowed him to use that command.

3.2.3 The selection of the groups.

A number of factors influenced the choice of the groups. Ideally, a selection based solely on the result of the programming aptitude test would have been preferred but as the students varied also in background, availability, etc., these factors also had to be taken into account.

Amongst other questions that appeared, the questionnaire given on day 1 asked the students whether they had any objections to taking part in a CAI course, whether they would be available until 1800 on days 2, 3 and 4 and whether they had had any previous programming experience, stating, if so, how much. No one at all objected to being taught by CAI methods so this did not affect the selection. However, three students indicated their non-availability to stay until 1800 (and hence be placed in group C) so they were placed two in group A and the other in group B. Four students indicated that they had had more than a week's programming experience and hence two

were placed in group A and one each in both of the other groups.

Out of the 22 students enrolled, four were girls and these were placed two in group B and one each in groups A and C. As mentioned before, 12 students were potential Computer Science candidates. These were divided equally. Of the remainder, four were placed in group A and three each in the other groups.

As the aptitude test was given on the first part of day 1 yet the groups had to be decided later that day in order to explain the different procedures involved for each group, there was a little difficulty involved in marking the aptitude tests in time. However the scores derived, even though some were later amended, formed the basis of the selection, after the constraints mentioned above had been imposed.

To discover whether there was any significant difference between the selected groups before the course, an analysis of variance on the corrected aptitude scores was carried out.

	Corrected Aptitude Scores.								Totals	Means
group A	77	94	64	91	16	88	49	70	549	68.6
group B	88	85	33	66	74	81	74		501	71.6
group C	92	69	55	99	56	85	74		530	75.7

Analysis of Variance Table.

source of variation	sum of squares	degrees of freedom	mean square	$F_{2,19}$
between treatments	188	2	94	<1
residual	8597	19	452	
total	8785	21		

3.2.4 The course log.

3.2.4.1 Day 1. Following a brief introduction in which CAI was only just mentioned, the students were given a pre-course attitude questionnaire. Then, the ICL programming aptitude test was given lasting about 90 minutes. Whilst the students were having a break, and then during the next short session which contained a brief talk on computing hardware, including slides, the students were selected for the different groups. After this they were told of their group assignments and the procedure to follow in each case.

3.2.4.2 Day 2. MTS became operational at 1400, at which time everyone in group B was told to commence. As at that time the students on the course were not only competing against each other for the loader but with every other user on the system, there was quite a delay before everyone got started. However, the last student had commenced by 1415.

At first the response was not as good as expected. This was owing to the fact that, even though the students were not using much CPU time, they were causing a lot of paging. The situation was greatly improved, however, when other users who currently had a large virtual memory were advised to leave their terminal and return after 1800 when the service would get better.

Group B were allowed to continue until 1530 when they were replaced by group C. As suspected, the first lesson was so long that no-one was finished in the allotted time. Only one person managed to get into the last section on program stops, four were at various stages of the section on running stored programs, whilst two were involved with parts and steps.

As the first examples class relied on the student's knowledge of parts and steps, the two students who had been unable to complete this section were given the necessary information by one of the demonstrators. However, both volunteered to return to a terminal after the examples session and, in fact, completed the lesson by the end of that day.

Despite the fact that the concept of CAI allows for self-paced instruction, it was decided that, through lack of time, the response files of some students should be updated overnight so that they would start on a par with the other students the next day. This was only carried out reluctantly and where essential, but when a student was skipped over any part of the instructional material he was furnished the next day with a copy of the conversation he might have had with that subject matter he missed. At the end of day 2, four group B students were updated in this manner.

The changeover to group C was effected in 10 minutes with the exception of one student, who, it was found out after some time, had got into a system disk error loop. She eventually commenced at 1615.

The response remained about the same until 1700 when it became progressively better owing to other users of the system going away. Even though the students were only required to stay until 1800, two stayed until 1820 and a further two until 1840.

Only one student managed to complete all four examples whereas three were in the middle of the third problem. These three were moved on to the start of the second lesson overnight. The remaining three, who were in various stages of the first example, were moved on to the last two examples. In all cases, sample output was given to these students whose response files were manually updated.

3.2.4.3 Day 3. In order to get the students started more quickly, it was arranged that only their terminals were to be activated for the first two minutes of the MTS session. This meant that they were able to load in NUTS much more quickly as they were only competing with one another and hence effectively gained from five to ten minutes. Everyone in group B commenced by 1410. The response was certainly much better than on day 2.

As it seemed more important to allow members of group B more time to complete their lesson than to enable members of group C to finish all of their problems, and, in any case, most of group C seemed willing to stay later than 1800, group B were given the use of the terminals until 1545. In spite of this, four students returned after their examples class as they wished to get through to the end of the second lesson. However, they were at no disadvantage during the class, as none of the examples involved work introduced towards the end of lesson two. By the end of day 3, three students had finished off the second lesson, two were almost finished the last part of it and the remaining two were updated overnight.

As in day 2, the changeover to group C took about ten minutes. Two students stayed until 1825 and another two until 1845. Again, only one student managed to finish off all four examples. Of the rest, one had completed three problems, two were in the middle of the second problem and the remaining three were struggling with the first one. Out of these, the latter five were moved on by two problems.

3.2.4.4 Day 4. The start of day 4 was much the same as that for day 3. The students were again given preference over other MTS users for about five minutes and this, coupled with the fact that they were becoming more proficient at signing on and restarting the course, meant that they were able to get started very soon after 1400.

The time for handing over to group C was made much more flexible on day 4, as it was the last lesson. Only one student finished the course without interruption, and that was by virtue of staying on until 1600 with the permission of the group C student who had finished all the examples in the previous days. Two more students returned after the examples class to finish off the final section on extended console I/O but the remaining four were simply given a copy of the subject matter they were unable to complete, the next day.

Of the students in group C, one had to leave at 1700 and hence was unable to commence the last set of examples, let alone complete them; two managed to finish off the four problems, but remained until 1850 to do so; one student was on each of problem 2 and 3, whilst the remaining two were struggling with the first problem. Copies of explanations and solutions to the first two examples of the last set were distributed as necessary on day 5.

3.2.4.5 Day 5. After the students were given a talk on the Computing Science Honours course, they received a short test on the programming language PIL. This was immediately followed by a post-course attitude

questionnaire for members of groups B and C whilst group A was given the opportunity to test on a terminal any of the examples they had written during the week. When group B had finished the questionnaire they were able to attempt examples on-line in the same manner.

Unfortunately, two students failed to appear for day 5 which was the last day of term. As a result, information regarding their post scores and post attitudes is unavailable and therefore affects some of the data described in 3.3.

3.2.4.6. In general. Over days 2, 3 and 4 as a whole, the following general observations were made.

There simply was not sufficient time to allow the majority of the students to complete the course at their leisure. One great advantage of CAI is that it provides individualised instruction, which, in itself, produces a self-paced situation. Consequently, a trade off had to be made so that the student was not deprived of any vital information. He was updated where necessary to the more important parts of the lessons while the missing material was provided in handouts.

Some students who considered the system quite slow in analysing responses and restarting after pauses did not pause when they were given the opportunity. They used the time taken for the restart to take effect after a pause to read the subject material.

A careful study of the students' response files was made for cases in which either the student was not given credit for a correct answer or he was not given assistance when he had made a common incorrect response. However only two cases of note were found and a later version corrected the anomolies.

Studying the students' response files also revealed that the response times for the questions asked in the section which describes boolean variables and expressions were much higher on average than any other section. This would suggest either that the material displayed was of poor quality or

that much more time should be spent on such a difficult aspect of a programming language.

3.3 Data obtained from the course and discussion of the results.

3.3.1 The Pre-test.

The pre-test given to the students was the ICL QUIS (Questions Using Instruction Sequences) programming aptitude test. A description of the test and its reliability followed by the reason for its choice in this context appear in this section.

3.3.1.1 The QUIS aptitude test. The test is designed to assess a student's ability to do programming work. It may also be valuable in helping to select operators since it is expected that an operator should at least be able to understand programming and to form simple programming sequences.

The test incorporates a number of problems which can be solved following the acquisition by a student of a carefully designed procedure and certain rules to be followed in presenting solutions. Thus, solutions must be presented in a clearly defined logical framework.

ICL feel that the test offers the following advantages over the normal type of intelligence test:

(i) a variety of response is permitted to each problem, to give better opportunity to the creative worker;

(ii) each problem requires a significant amount of thought and analysis and it is impossible to provide a correct solution by luck;

(iii) the test is not only of ability to solve problems but also of ability to comprehend and apply a simple 'programming language'; and

(iv) since each solution has to be developed and presented in a number of sequential steps, the solution reveals not only whether it is correct or incorrect but also shows how errors have been made and also the economy of correct solutions; thus, it is possible to use a marking system which can award marks for solutions of various quality.

The procedure for giving the test is as follows. Students are given a booklet which describes the nature of the problems and explains how solutions are to be presented, emphasising the special rules which have to be followed. Examples are then worked by the students to familiarise themselves with the technique. The actual test contains ten problems and one hour is allowed for it.

3.3.1.2 Reliability of the test. For several years, students who have taken the Joint Honours course which includes the Computing option have been given the QUIS test at the beginning of the course. Their results have since been compared with their eventual showing in the examination at the end of the year. As the aptitude test is designed to test ability to do programming work, the scores were compared with the results of the practical examination only. The scores are available for the last two years and exclude those students who have either seen the test before or are repeating the year. Exactly one hundred student scores are to hand and are included in the following analysis (see Figure 3.2).

Analysis of Variance Table.

source of variation	Sum of squares	Degrees of freedom	mean square	Fvalue
due to regression	789	1	789	11.3**
2 parallel lines versus 1 single line (between years)	793	1	793	11.3**
residual after fitting 2 parallel lines	6788	97	70	/
total sum of squares	8370	99	/	/

The tabulated F value on 1 and 97 degrees of freedom is 7.0 at the 1% level. Of lesser importance is the fact that there is a significant difference in performance between the two years. The main conclusion is that the dependence of a student's practical examination score on his programming aptitude score is significant at the 1% level, which suggests that the test may be an estimate of programming ability.

Graph of practical examination score against QUIS score for Joint Honours students.

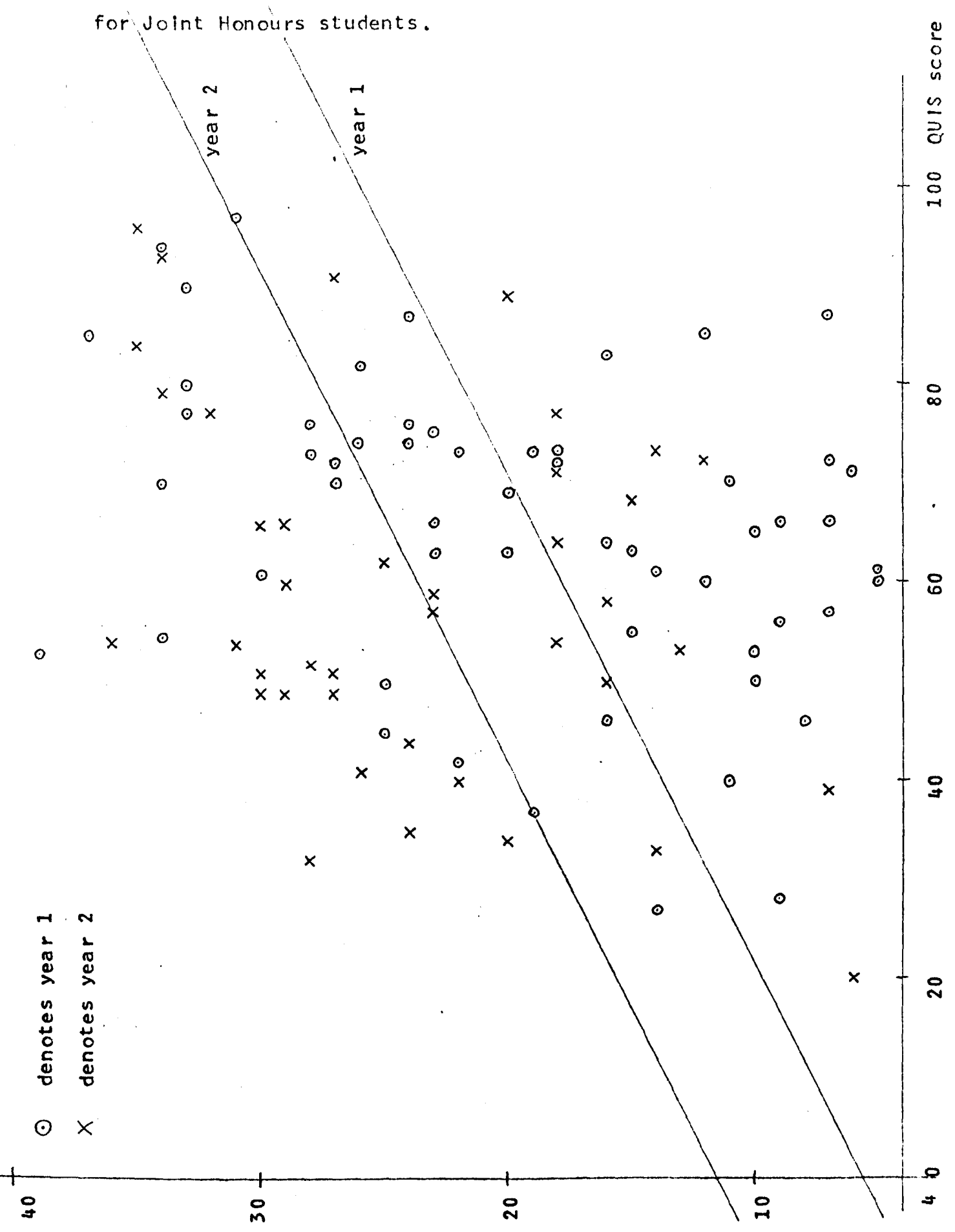


Figure 3.2

3.3.1.3 Choice of the test for this course. The results quoted above for the reliability of the QUIS test are based on observations taken at an interval of nine months. During this time, how hard the student worked on his programming problems contributed to his examination score. In the case of the PIL course, the post-test was to be given in the same week as the QUIS test so that such factors as loss of interest and interference from other sources would not affect the validity of the test as much. This fact, together with the fact that the course was in particular about a programming language suggested strongly that the QUIS test was a good choice in this situation.

3.3.2 The post-test.

The post-test was designed with the thought in mind that if it were either too easy or too difficult then not a sufficiently large spread would be obtained in the scores. Also, the test had to determine the students' comprehension of what they had been taught during the week as well as their ability to understand simple programming techniques. To this end, the categories of questions that were considered necessary were factual questions, expression evaluation, simple program tracing and simple program writing.

The actual questions that appeared are contained in Appendix B, together with the marking scheme adopted. Thus, the marks allotted to the four categories mentioned above are 18, 38, 16 and 28, respectively.

3.3.3 Analysis of the scores for the pre-test and the post-test.

We are concerned with the use of the analysis of variance technique to compare the post-test scores for the three groups, the dependent variables, y , when the effects of differences in the values of some underlying, independent variable, x , the pre-test score, have been eliminated. The object is to allow the comparisons between the values of the dependent variable to be made more accurately. In other words, comparisons of the effects of placement in a particular group on the post

score of a student may be made more accurately if the effects of unequal initial aptitude score are eliminated. Measurements of this type which may be used to account for some of the variability in y have been called 'concomitant' observations. It is important that the values of concomitant variablesare not affected by the treatments which the experiment is designed to compare. Obviously, in our case, this is satisfied.

The basic assumption is that within each homogeneous group there is a linear regression of y on x with the same slope and with normally distributed errors of constant and equal variance for all groups.

The following table gives the scores obtained for both tests. A linear transformation has been applied to the pre-test score. This will not affect any calculations but as the transformation has the result of giving the pre-test scores the same mean as the post-test scores then the effect is that comparisons of the groups and of particular students "by eye" are made far easier. Graphs of post-test score against pre-test score for groups A, B and C are contained in figures 3.3, 3.4 and 3.5, respectively.

Two students failed to appear for the post-test and hence are not contained in the analysis. Their respective aptitude scores (transformed) were 43 for the missing group A student and 64 for the missing group B student.

GROUP		A		B		C	
Variable		y(post score)	x(pre score)	y	x	y	x
Observation		73	67	43	77	57	80
		79	82	57	74	70	60
		61	56	77	29	73	48
		67	79	58	57	51	86
		34	14	90	70	66	49
		66	77	40	64	74	74
		41	61			90	64
Totals	$T_{y,j}$	421		365		481	
	$T_{x,j}$		436		371		461
	n_j		7		6		7

Total number of observations, N = 20. Number of treatments, k = 3.

Graph of post-test score against pre-test score for group A.

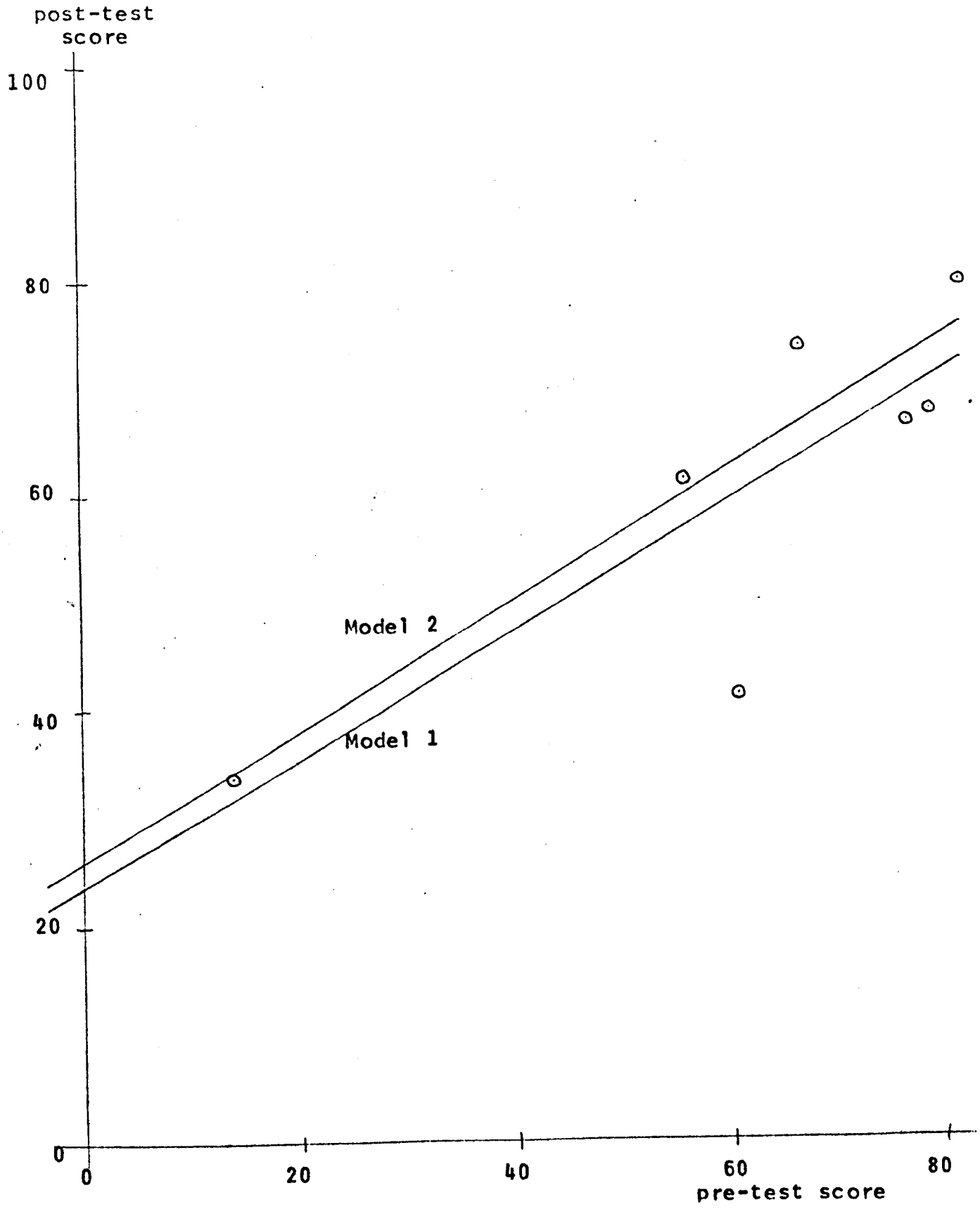


Figure 3.3

Graph of post-test score against pre-test score for group B.

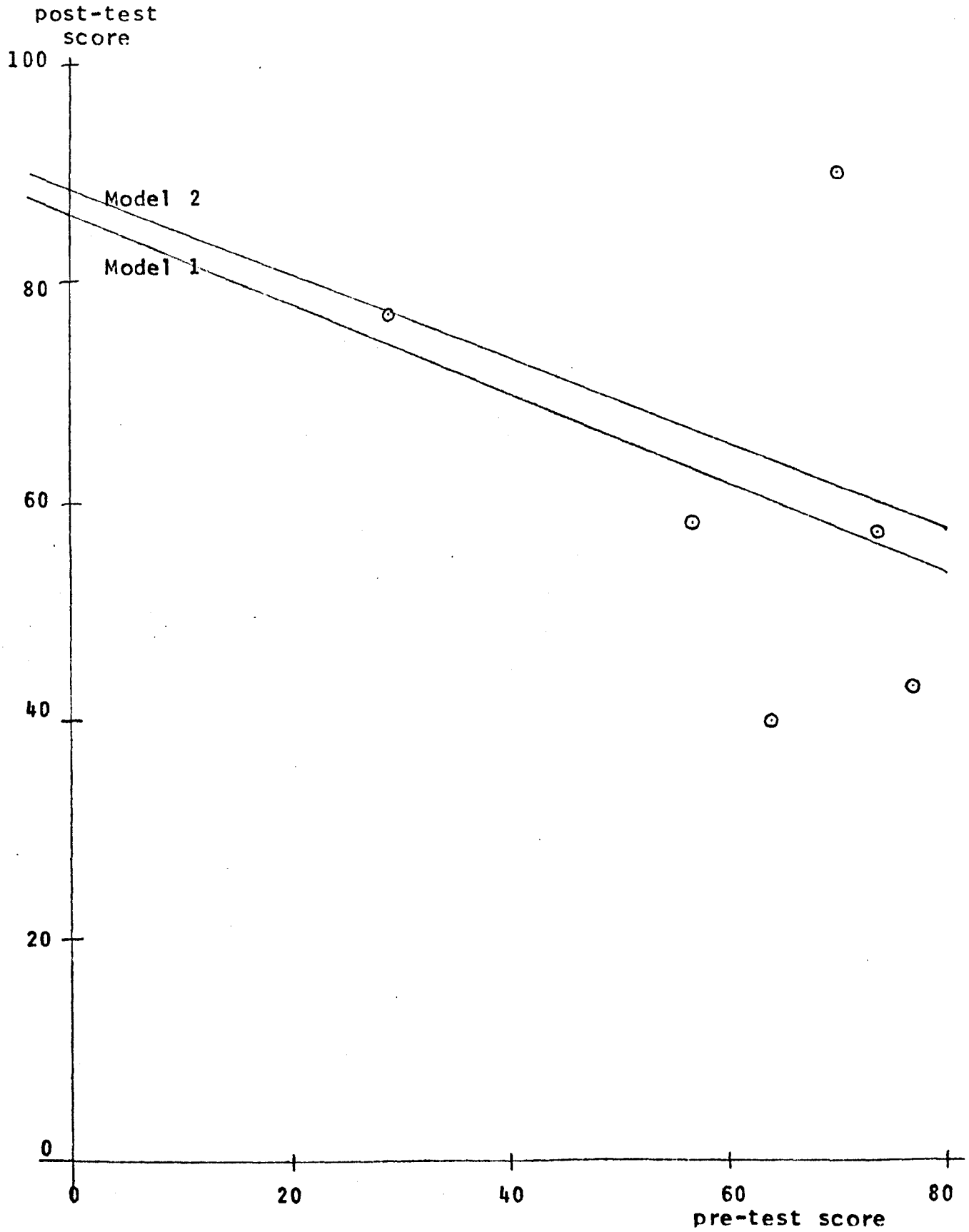


Figure 3.4

Graph of post-test score against pre-test score for group C.

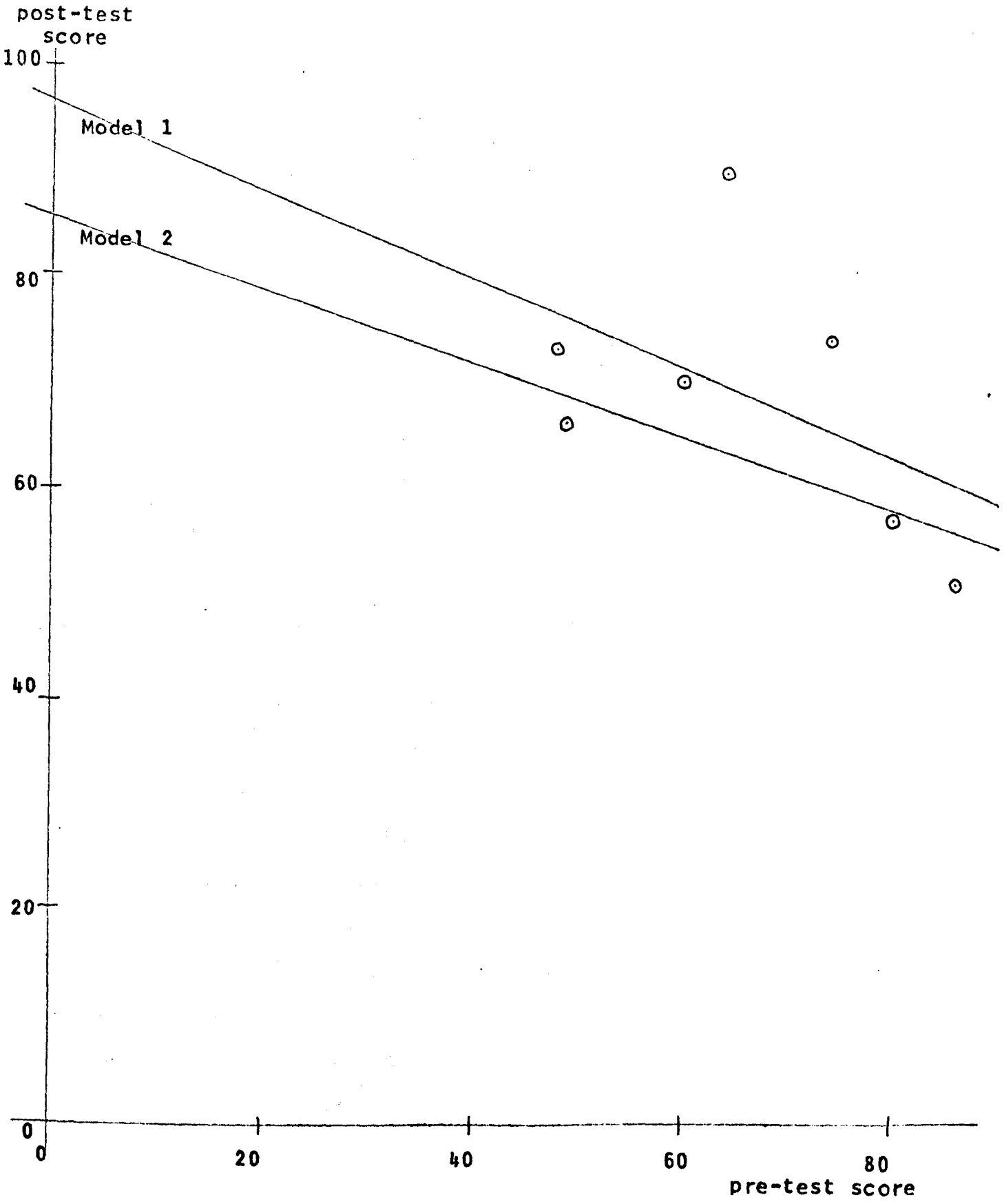


Figure 3.5

The following statistics were computed.

	A	B	C	total
Σy_i^2	26973	24091	34011	85075
Σy_i	421	365	481	1267
$(y^2)_j$	1653	1887	959	4499
Σx_i^2	30436	24491	31673	86600
Σx_i	436	371	461	1268
$(x^2)_j$	3279	1551	1313	6143
$\Sigma x_i y_i$	28137	21928	31120	81185
$(xy)_j$	1915	-641	-557	717
$\hat{\beta}_j$	0.5840	-0.4133	-0.4242	/
$(xy)_j^2/(x^2)_j$	1118	265	236	1619
τ	0.8226	-0.3746	-0.4964	/

Residual sum of squares after fitting 3 slopes, $R(\hat{\underline{\alpha}}, \hat{\underline{\beta}})$

$= 4499 - 1619$ with $(N-2k)$ d.f. $= 2880$ with 14 d.f.

Residual sum of squares after fitting 3 parallel lines, $R(\hat{\underline{\alpha}}, \hat{\beta})$

$= 4499 - \frac{717^2}{6143}$ with $(N-k-1)$ d.f. $= 4415$ with 16d.f.

These give the following analysis of covariance table.

source of variation	degrees of freedom	sum of squares	mean square	Fvalu
3 different v 1 common slope (within groups)	2	1535	768	3.7
residual after fitting 3 slopes (within groups)	14	2880	206	/
residual after fitting 3 parallel lines	16	4415	/	/

Tabulated $F_{2,14}$ at the 5% level is 3.7. Thus, as the derived F-statistic is equal to the tabulated value at the 5% level, we may reject the null hypothesis that within each group there is a linear regression of y on x with the same slope and suggest that the treatment effects differ significantly. The fitted regression equations are:

group A : $y = 0.5840x + 23.76$

group B : $y = -0.4133x + 86.38$

group C : $y = -0.4242x + 96.65$

Standardising at the most convenient pre-test score, 63.4, the overall mean, the adjusted estimated post-test scores become

group A : 60.8

group B : 60.2

group C : 69.8

From the fitted slopes, it appears that the difference between groups is caused by group A exhibiting a different effect from both group B and group C. Standardising at the pre-test mean, the post score for group C appears higher than those for groups A and B but as the slopes are so different we must introduce a different model to attempt to compare the group effects further.

In the above model we have

$$E\{y_{ij}|x\} = \alpha_j + \beta_j(x-\bar{x}) \qquad \text{for } j = 1, 2, 3.$$

This gave a residual with 14 degrees of freedom as we estimated 6 parameters.

The second model is

$$E\{y_{ij}|x\} = \gamma + \delta_j(x-\bar{x}) \qquad \text{for } j = 1, 2, 3.$$

In other words, at the overall mean, the intercept is the same for each group and the residual will have 16 degrees of freedom as we are estimating 4 parameters.

Residual sum of squares after fitting 3 slopes with a common intercept, $R(\hat{\gamma}, \hat{\delta})$ = 3261 on 16 d.f., which gives an estimated variance of 204. Adding a further line to the analysis of covariance table we get that for 3 different versus 1 common intercept (within groups), mean square = 190 and degrees of freedom = 2. This estimate does not give a significant result.

From this analysis we may conclude that the positive response given by group A differs significantly at the 5% level from the negative responses given by both group B and group C.

We might calculate the estimate of the missing score from group B. The second model gave:

$$\hat{\gamma} = 63.69 \quad , \text{var}\{\hat{\gamma}\} = 10.34$$

$$\hat{\delta}_1 = 0.5907 \quad , \text{var}\{\hat{\delta}_1\} = 0.0620$$

$$\hat{\delta}_2 = -0.3924 \quad , \text{var}\{\hat{\delta}_2\} = 0.1306$$

$$\hat{\delta}_3 = -0.3475 \quad , \text{var}\{\hat{\delta}_3\} = 0.1521$$

Thus, $\hat{E}\{y|\ x = 64, \text{group B}\} = 63.46$

and the 95% confidence interval is (56.17.70.75).

3.3.4 Responses and their relation to performance.

Examination of the students' response files returns totals for the number of questions the students from groups B and C answered during the CAI lessons and the number of these questions that were answered correctly first time. The main reason why different students answered differing numbers of questions has been mentioned previously. It is that there was insufficient time to allow all students to come to completion at their own pace. In addition, some students took slightly different routes through the course and in some cases this produced a different number of questions.

From this data, it was hoped to show that there was no evidence to support the assumption that students who had been in a position to attempt more questions, and hence seem more of the course actually on a terminal, might do better in the post-test. In other words it was hoped to determine whether the handouts given when the students were unable to complete the lesson had had the desired effect of suitably replacing the conversation for which there was insufficient time.

Also, it was hoped to find out if a relationship existed between post-test score and percentage of questions answered correctly at the first attempt.

The observations were:

post-test score	number of questions attempted	% right first time
43	49	57.1
57	68	63.2
77	63	61.9
58	47	63.8
90	53	67.9
40	48	41.7
57	59	59.3
70	74	66.2
73	87	57.5
51	89	69.7
66	80	57.5
74	61	63.9
90	89	69.7

A regression analysis was performed against both the other variables in turn. The observations were treated as coming from one source, not two

separate groups as, no matter in what group the student was placed, he still received identical CAI lessons. Graphs of the post-test score against the number of questions attempted and the percentage right first time are contained in figures 3.6 and 3.7 respectively.

The analysis of variance table for the regression of post-test score against the number of questions attempted is:

source of variation	degree of freedom	sum of squares	mean square	$F_{1,11}$
due to regression	1	380	380	1.6
deviation about regression	11	2667	242	/
total	12	3047	/	/

The tabulated value of $F_{1,11}$ at the 5% level is 4.8. Thus, we may conclude that there is no significant dependence of the post-test score on the number of questions attempted. This suggests that the handing out of simulated dialogues to some students owing to insufficient time did not adversely affect those students' performance in the post-test.

The analysis of variance table for the regression of post-test score against the percentage of questions answered correctly at the first attempt is

source of variation	degrees of freedom	sum of squares	mean square	$F_{1,11}$
due to regression	1	1079	1079	6.0
deviation about regression	11	1968	179	/
total	12	3042	/	/

The equation of the fitted line is $y = 1.279x - 13.57$;

estimated S.E. $\{\hat{\beta}\}$ = 0.5207, 95% confidence interval for β is (0.133, 2.425);

correlation coefficient, r = 0.595.

As the F value exceeds the tabulated $F_{1,11}$ statistic at the 5% level, we may conclude that the post-test score may be estimated from the percentage of questions answered correctly first time using the above linear relationship.

Using this to estimate the missing group B post score, we have:

$\hat{E}\{y|x=59.7\} = 62.79;$

Graph of post-test score against percentage of questions answered correctly first time.

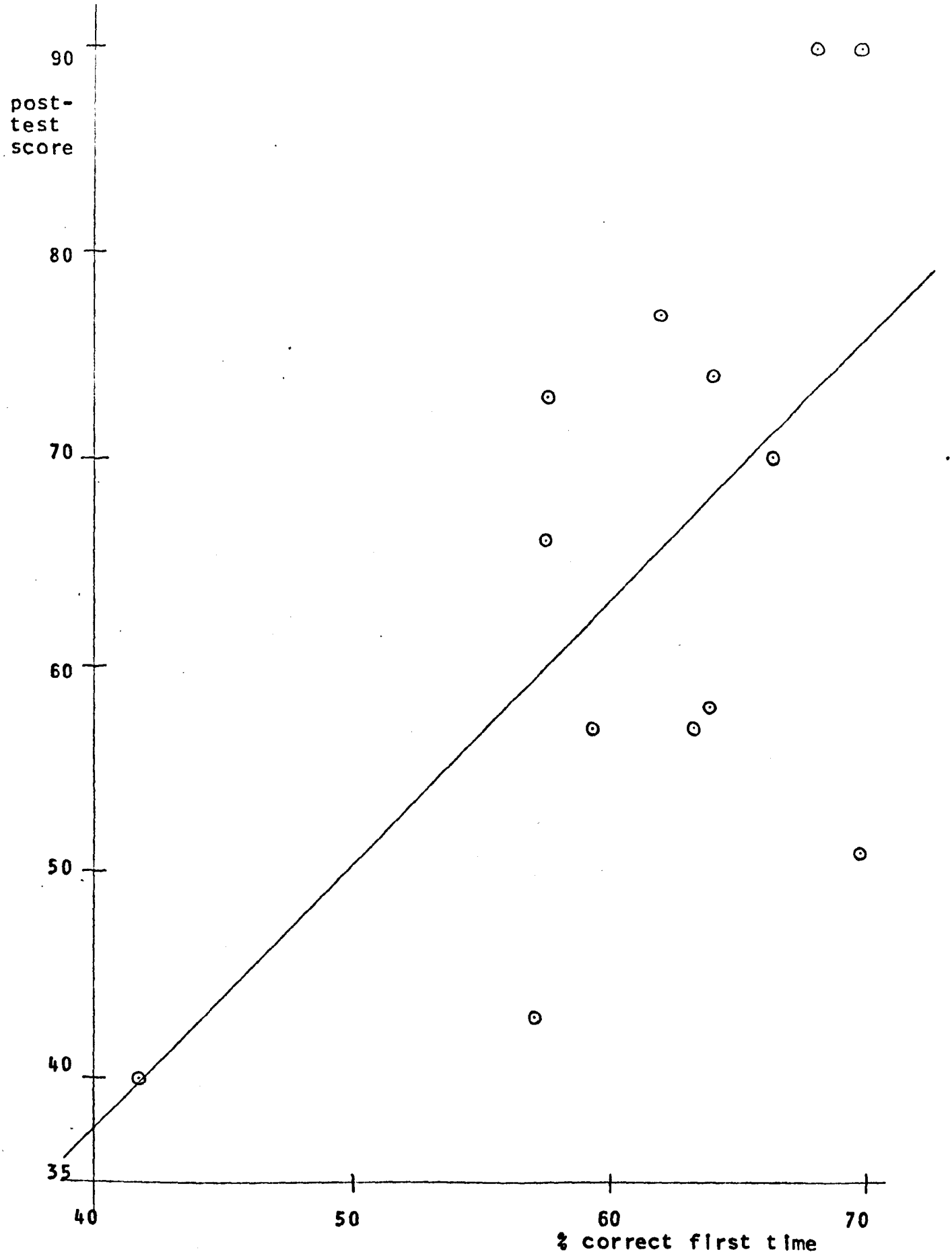


Figure 3.6

Graph of post-test score against number of questions answered during the CAI lessons.

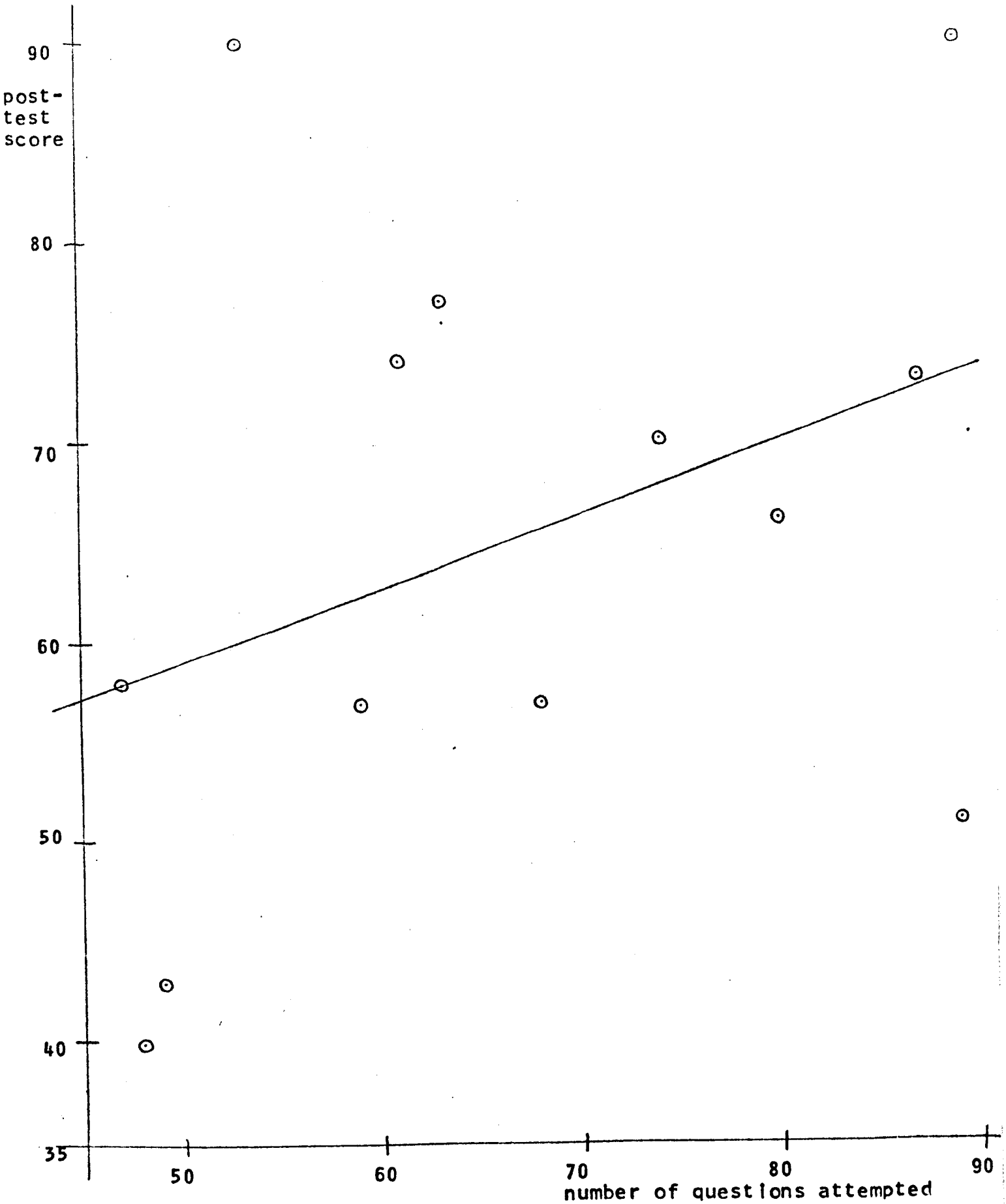


Figure 3.7

estimated S.E. $\{\hat{E}(y|x=59.7)\} = 3.77$;

and the 95% confidence interval is (54.49,71.09).

3.3.5 Response times and their relation to performance.

In addition to information about the number of questions attempted, a search through the students' response files also furnishes data on the time for every response and how many responses were made (as opposed to how many questions were set). From this data, it was hoped to ascertain whether a relationship existed between post-test score and any one or more of:

(i) average response time for the first attempt;

(ii) average response time over all responses;

(iii) average response time per question, summing times over all attempts; and

(iv) ,(v) and (vi) their inverses.

The observations were:

post-test score	no. of questions attempted	no. of attempts made	total time (sec)	time for first responses (sec)
43	49	65	4426	3242
57	68	92	5984	4913
77	63	84	6905	4982
58	47	75	5325	2680
90	53	72	4274	2396
40	48	82	7418	4510
57	59	76	6631	5128
70	74	90	7771	6653
73	87	122	6188	5115
51	89	113	8082	6715
66	80	102	4199	3447
74	61	83	6158	4599
90	89	114	4454	3821

From this table, a regression analysis was performed on the post-test score against each of the six variables tabulated below.

post-test score	av. time for first response (sec)	recip. of av. time for first response ($\times 10^{-2}$ sec^{-1})	av. time for all responses (sec)	recip. of av. time for all responses ($\times 10^{-2}$ sec^{-1})	av. time per question (sec)	recip. of av. time per question ($\times 10^{-2}$ sec^{-1})
43	66.2	151	68.1	147	90.3	111
57	72.3	138	65.0	154	88.0	114
77	79.1	126	82.2	122	109.6	91
58	57.0	175	71.0	141	113.3	88
90	45.2	221	59.4	168	80.6	127
40	94.0	106	90.5	111	154.5	65
57	86.9	115	87.3	115	112.4	89
70	89.9	111	86.3	116	105.0	95
73	58.8	170	50.7	197	71.1	141
51	75.4	133	71.5	140	90.8	110
66	43.1	232	41.2	243	52.5	191
74	75.4	133	74.2	135	101.0	99
90	42.9	233	39.1	256	50.0	200

Graphs of the post-test score against each of these six variables in turn are contained in figures 3.8, 3.9, 3.10, 3.11, 3.12, and 3.13.

3.3.5.1 Regression of post-test score against the average time for the first response to a question.

source of variation	degrees of freedom	sum of squares	mean square	$F_{1,11}$
due to regression	1	875	875	4.4
deviation about regression	11	2172	197	/
total	12	3047	/	/

As the tabulated value of $F_{1,11}$ at the 5% level is 4.8, the result is not significant. Hence, we may conclude that there is a suggestion from the data that there is no linear relationship between the post-test score and the average time that the student took to make his first response.

3.3.5.2 Regression of post-test score against the reciprocal of the average time for the first response to a question.

source of variation	degrees of freedom	sum of squares	mean square	$F_{1,11}$
due to regression	1	987	987	5.3
deviation about regression	11	2060	187	/
total	12	3047	/	/

Graph of post-test score against average time for first response.

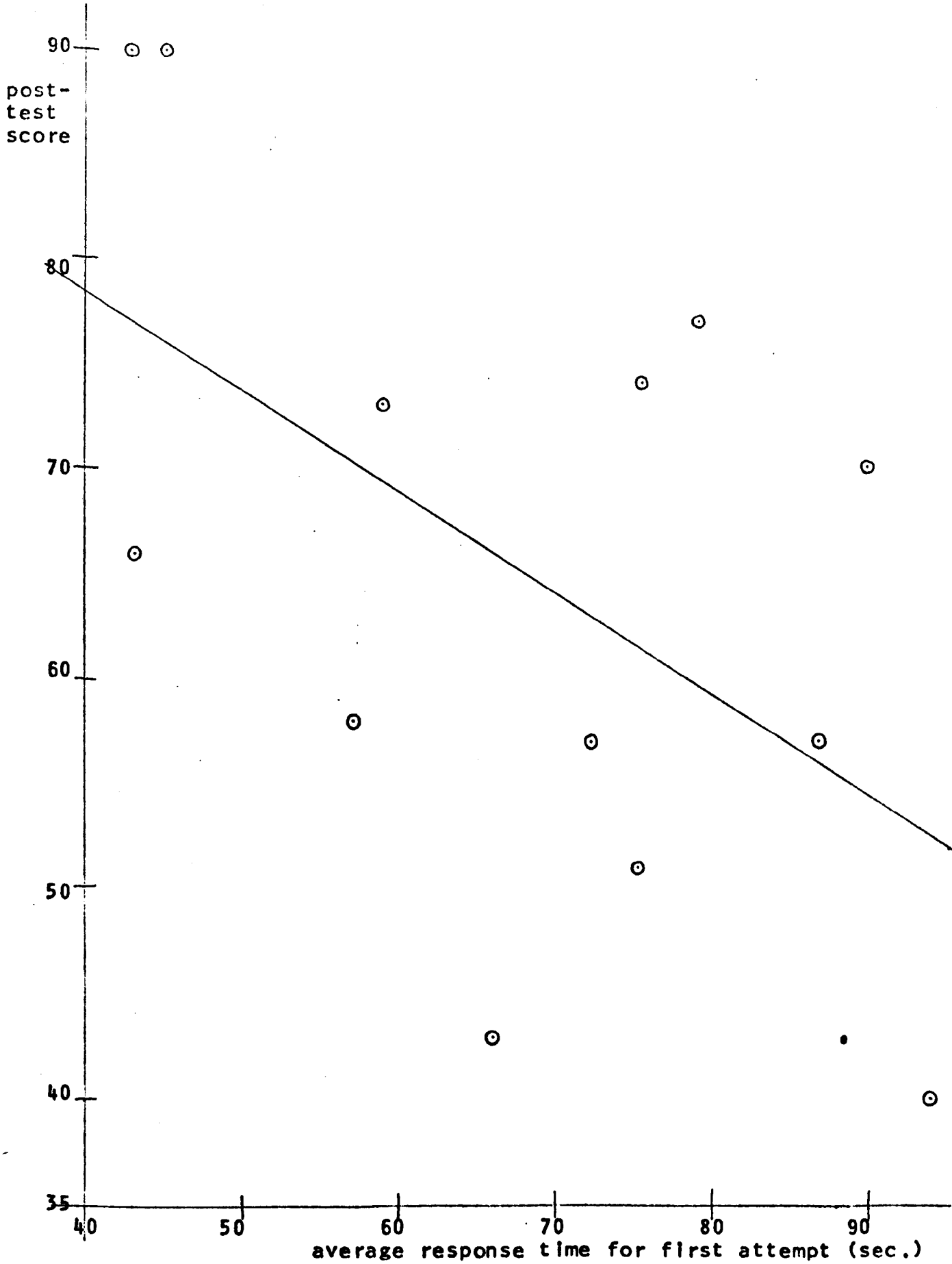


Figure 3.8

Graph of post-test score against reciprocal of average time for first response.

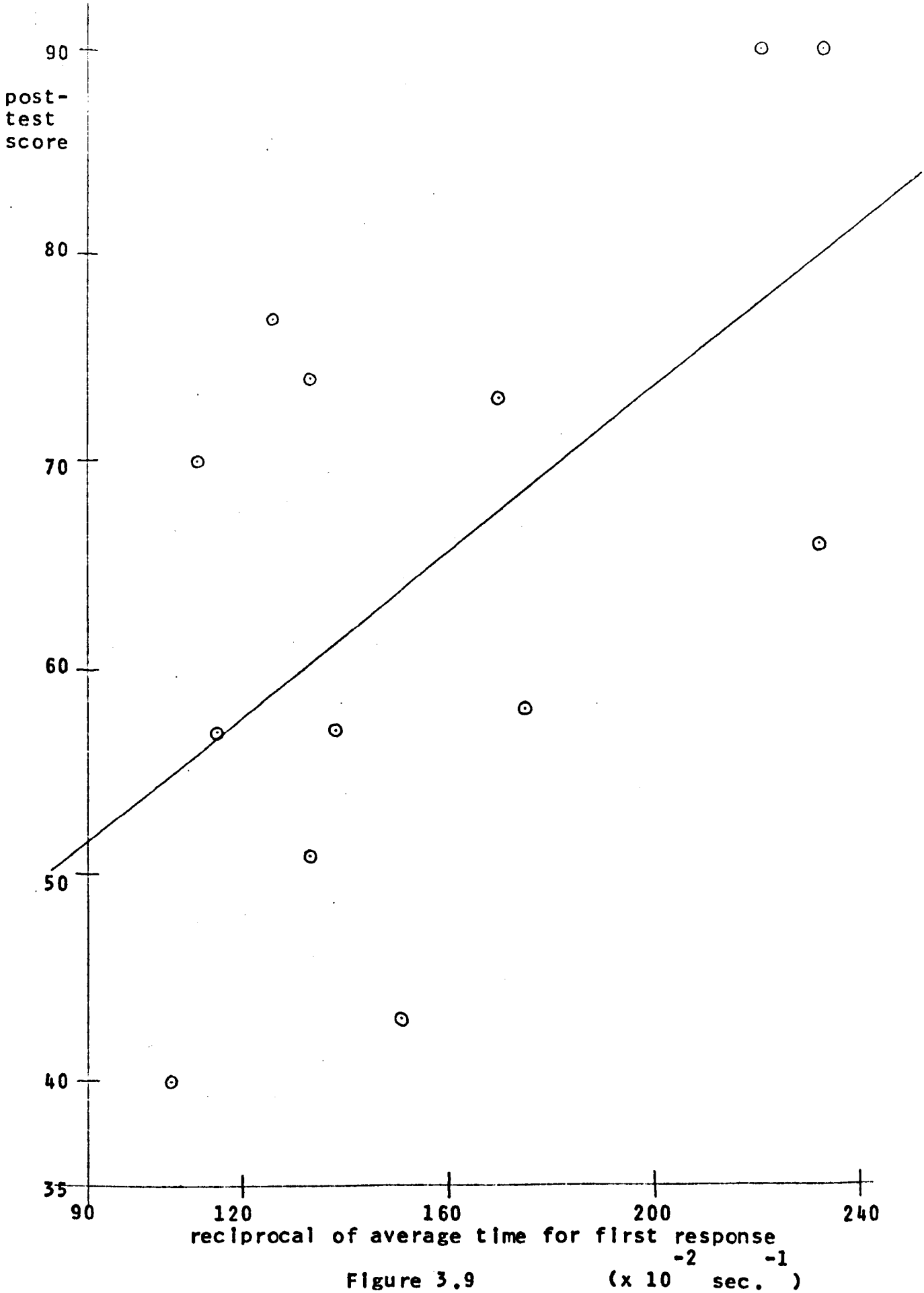


Figure 3.9

Graph of post-test score against average time over all responses.

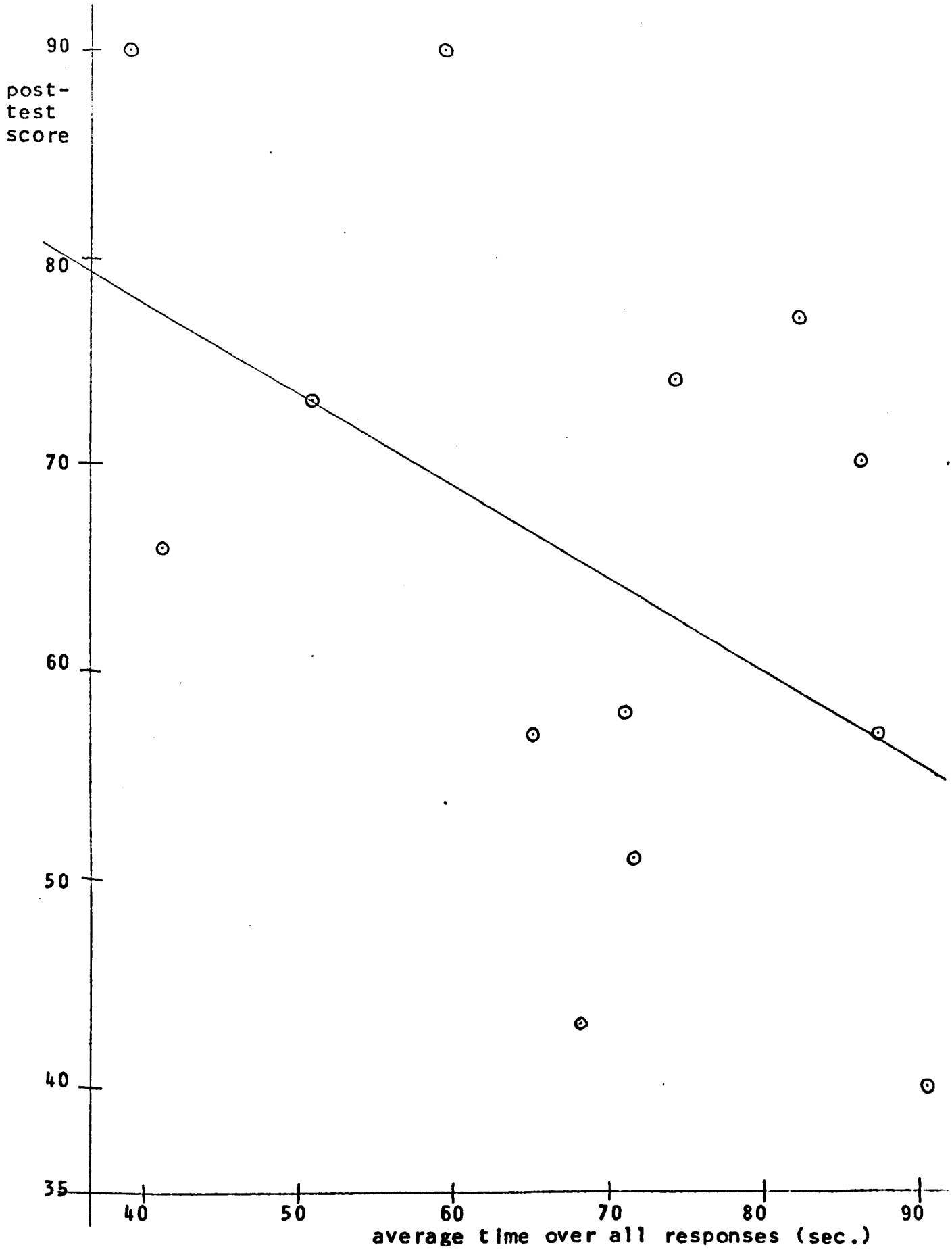


Figure 3.10

Graph of post-test score against reciprocal of the average time taken over all responses.

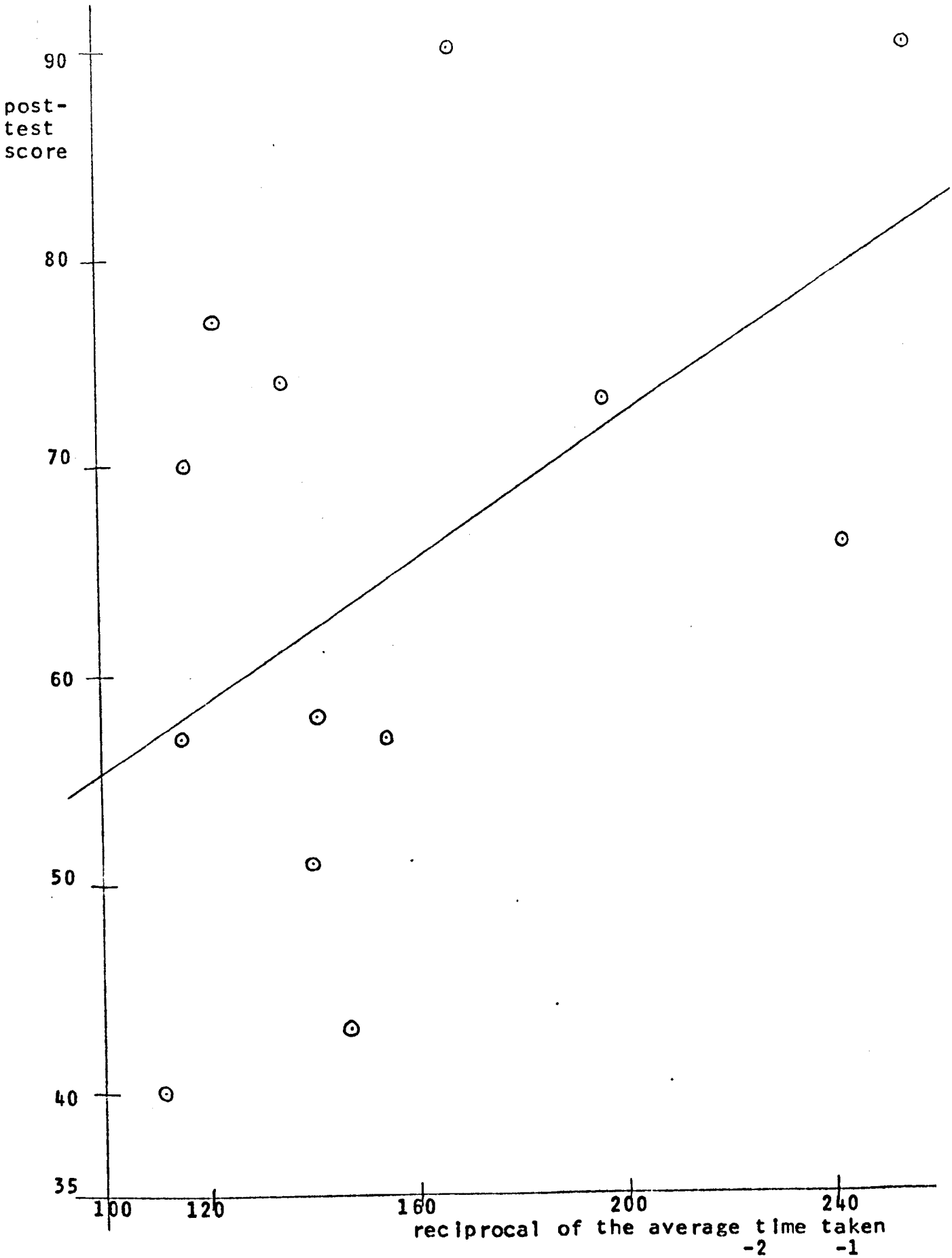


Figure 3.11

Graph of post-test score against average total response time per question answered.

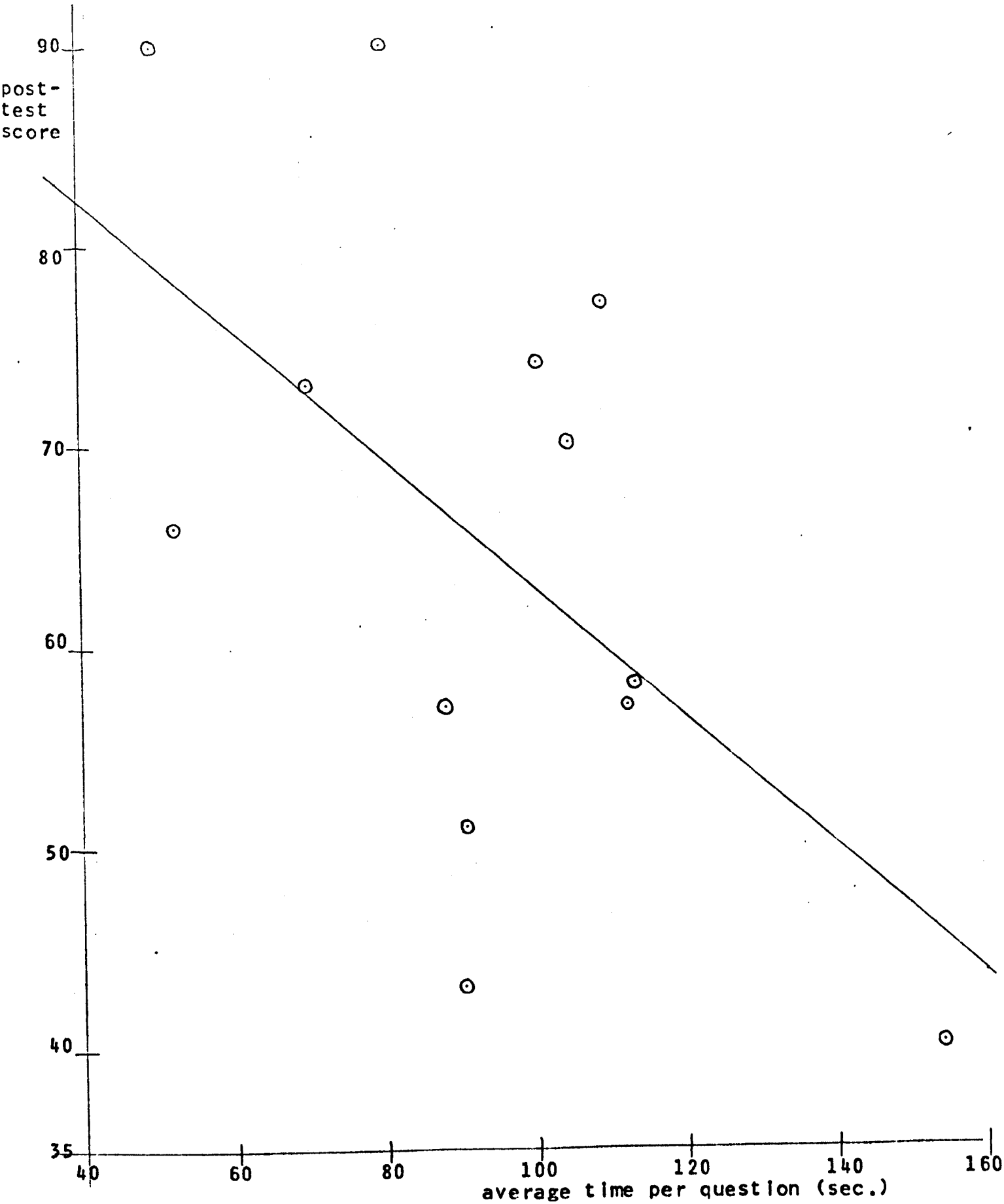


Figure 3.12

Graph of post-test score against the reciprocal of the average total response time per question answered.

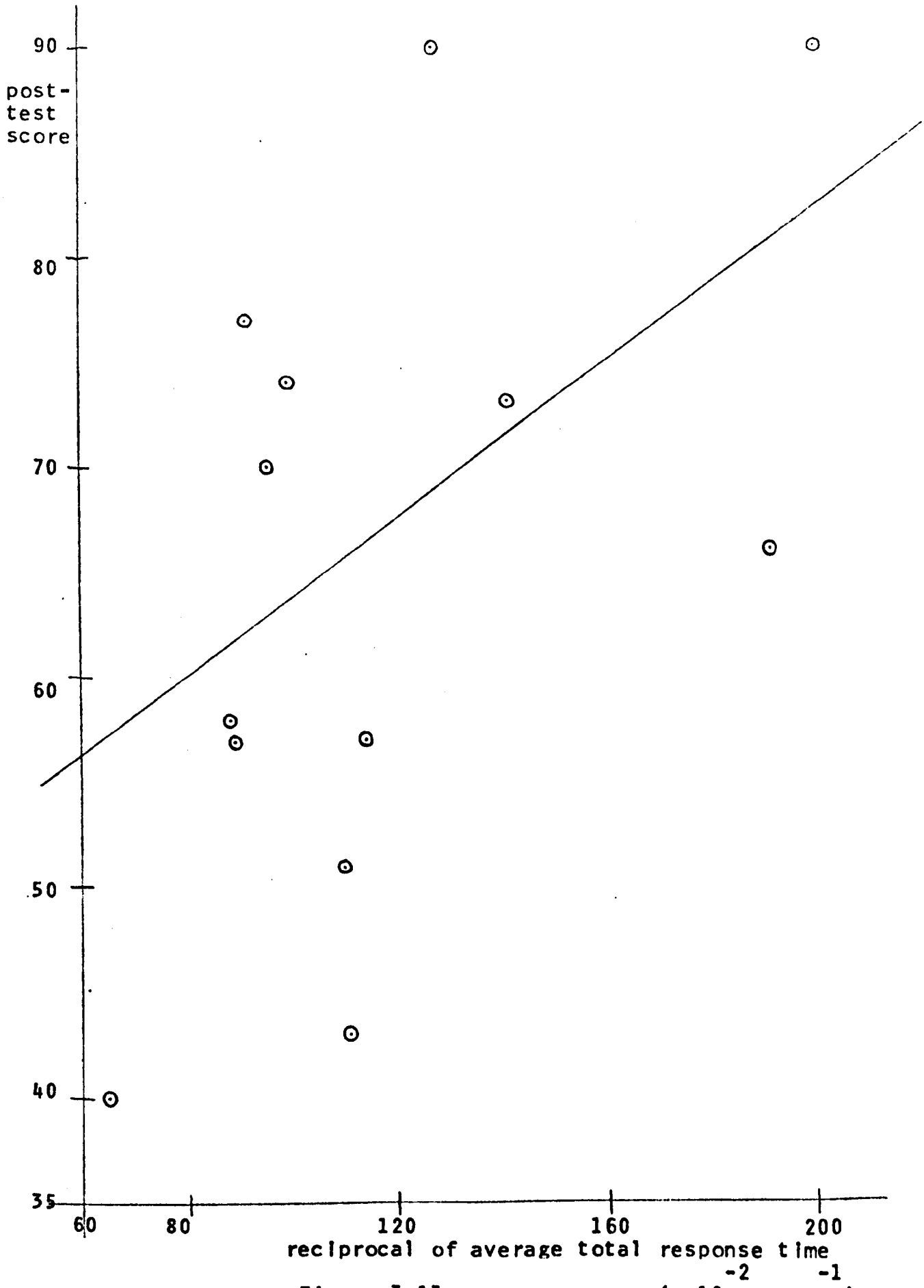


Figure 3.13

The fact that the F value obtained is significant at the 5% level tends to suggest that the post-test score may be inversely proportional to the average time a student takes to make his first attempt.

The equation of the fitted line is $y = 0.1989x + 33.81$;

estimated S.E. $\{\beta\} = 0.0866$; 95% confidence interval for β is (0.0084, 0.3894);

correlation coefficient, $r = 0.569$.

Using the above relationship to estimate the missing group B post score, we have:

$\hat{E}\{y|x=154\} = 64.44$; estimated S.E. $\{\hat{E}(y|x=154)\} = 3.81$;

95% confidence interval is (56.07, 72.81).

3.3.5.3 Regression of post-test score against the average time taken over all responses made.

source of variation	degrees of freedom	sum of squares	mean square	$F_{1,11}$
due to regression	1	705	705	3.3
deviation about regression	11	2342	213	/
total	12	3047	/	/

The F value derived is not significant. Thus, the data suggests that no linear relationship exists between the post-test score and the average response time taken over all responses made.

3.3.5.4 Regression of post-test score against the reciprocal of the average time taken over all responses made.

source of variation	degrees of freedom	sum of squares	mean square	$F_{1,11}$
due to regression	1	726	726	3.4
deviation about regression	11	2321	211	/
total	12	3047	/	/

The F value obtained is not significant. Hence, the data suggests that the post-test score is not inversely proportional to the average response time taken over all responses made.

3.3.5.5 Regression of post-test score against the average total response time per question answered.

source of variation	degrees of freedom	sum of squares	mean square	$F_{1,11}$
due to regression	1	961	961	5.1
deviation about regression	11	2086	190	/
total	12	3047	/	/

The derived F value is significant at the 5% level. This tends to suggest that the post-score may be directly proportional to the average total response time per question answered, with negative gradient.

The equation of the fitted line is $y = -0.3231x + 95.38$; estimated S.E. $\{\hat{\beta}\} = 0.1437$; 95% confidence interval for β is (-0.6392, -0.0070); correlation coefficient, $r = -0.562$

Using the fitted parameters to estimate the missing group B post score, we have:

$\hat{E}\{y|x=78.7\} = 69.95$; estimated S.E. $\{\hat{E}(y|x=78.7)\} = 4.39$;

95% confidence interval is (60.29,79.61).

3.3.5.6 Regression of post-test score against the reciprocal of the average total response time per question answered.

source of variation	degrees of freedom	sum of squares	mean square	$F_{1,11}$
due to regression	1	810	810	4.0
deviation about regression	11	2237	203	/
total	12	3047	/	/

The F value obtained is not significant. Hence, the data suggests that no inversely linear relationship exists between the post-test score and the total response time per question.

3.3.6 Performance of group C students during examples classes.

Section 3.2.2 outlines the structure of the examples classes which group C students were given on terminals. The students' response files contain all the information about which choice they made at any of the decision stages. Of particular interest are the numbers of times the INFO and HELP facilities were used during solution of the problems.

The following table shows for each student:

(i) the fraction of the number of questions he answered in which he requested INFO (maximum 1.00);

(ii) the average number of HELP elements he received per answered question; and

(iii) the fraction of the number of questions he answered correctly/incorrectly with/without HELP.

	INFO requests	HELP elements	correct with no help	correct with help	wrong with no help	wrong with help
	0.50	0.75	0.25	0.25	0.25	0.25
	0.75	1.00	0.50	0	0.25	0.25
	0.67	0.83	0.33	0	0.33	0.33
	0.29	0.43	0.86	0	0	0.14
	0.43	0.86	0.71	0.29	0	0
	1.00	3.25	0	0.75	0	0.25
	0.50	0.41	0.83	0.17	0	0
overall	0.50	0.89	0.59	0.18	0.09	0.14

The discouraging part of this data is that on average INFO was requested for only 50% of the questions attempted by all students and the average number of HELP elements accessed was 0.89 out of, on average, 8 such elements per question. However, this may be explained by the fact that 59% of all questions attempted were answered correctly without any assistance. Of the 41% remaining, 18% were answered correctly after assistance but only 9% were answered incorrectly without any help at all. This 9% is made up by 3 students only.

An attempt was made to give an example class score to each of the students, based on such items as whether they answered questions correctly, whether they asked for HELP or INFO, and whether they made more than the minimum number of attempts required. The rationale for deciding the scoring scheme was as follows. For a correct response:

(i) with no assistance, 20 marks;

(ii) if INFO was requested then 4 marks were deducted;

(iii) if HELP was requested then 2 marks per elements were deducted;

(iv) so long as the response was eventually correct, the student was not penalised for extra attempts; and

(v) a floor was placed on the score for a correct answer, 10 marks -

this meant that only the first three HELP elements warrant a deduction.

For an incorrect answer:

(i) each additional attempt over the minimum of two gained 2 marks;

(ii) a request for INFO gained 3 marks;

(iii) each HELP element added 1 mark; and

(iv) a ceiling of 9 was imposed which meant that only the first four HELP elements affected the score.

This scheme was applied to each question attempted and the average over the number of questions, converted to a percentage, was attributed to each student. The scores were:

post-test score	fitted examples class score
57	49
70	60
73	50
51	85
66	86
74	48
90	83

The correlation coefficient for these scores is -0.0001. Consequently, we may conclude that the examples class scores that were fitted have no bearing whatsoever on post-test scores. A scatter diagram is shown in figure 3.14.

3.3.7 The attitude questionnaires.

Much information on the student's ability and performance is given by the results of the programming aptitude test and the post-test and by examination of the students' response files for the course. However, to gain such valuable data as the students' attitudes to the course, both beforehand and afterwards, their change of opinion on certain aspects over the week and their assessment of the structure, content and

Scatter diagram of post-test score against fitted examples class score.

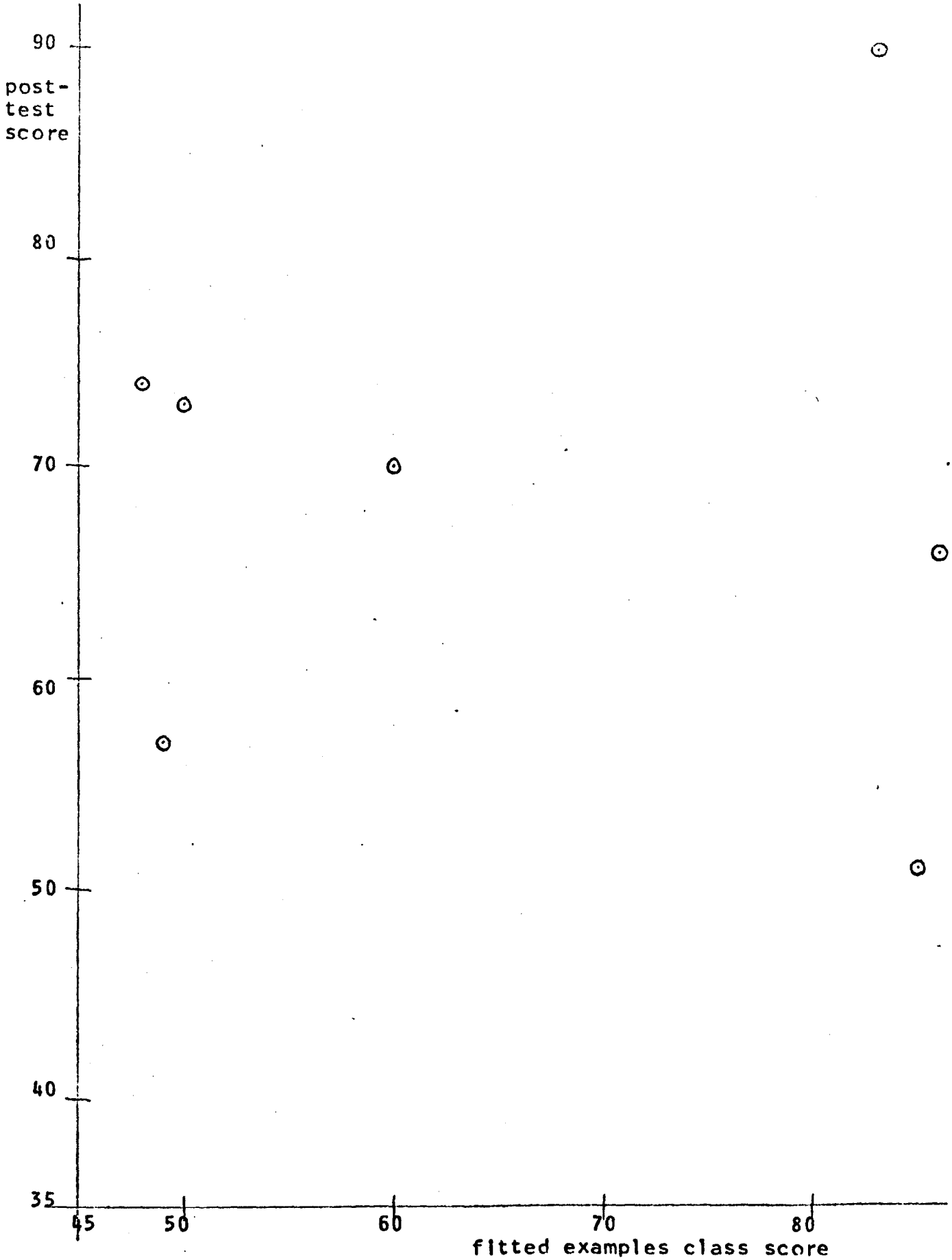


Figure 3.14

presentation of the course, two questionnaires were given. Unfortunately, owing to lack of suitable sets of subjects and time before the course itself, it was not possible to arrange a pre-trial for the questionnaires.

The first questionnaire was given immediately on commencement of the course on day 1. All that the students were told was that some of them were to receive the course from a typewriter terminal - they were not "sold" the advantages of CAI. Also included in this questionnaire were such questions from which selection of the groups could be made.

The second questionnaire was given on day 5 after the instruction had been completed. Only groups B and C were given it as the questions were not applicable to those taught conventionally since the questionnaire assumed the experience of a CAI course.

The results of the questionnaires are contained in Appendix C. For questions which seek a comparison of attitudes before and after, the response matrices follow the two questionnaires. The results are divided into sub-totals for each group. For the pre-questionnaire, the grand total includes the answers recorded for the two students who did not attend day 5.

3.3.7.1 Pre-course attitude.

Before the course, the students were asked their opinions on various aspects of their impending use of terminals for the course. The vast majority regarded their typing as being too slow yet about the same number decided that they would not be inconvenienced in having to wait for the course material to be typed out. Similarly, most of them thought that listening to a typewriter for an hour or so would do their mental health no harm at all. It was no surprise, therefore, that only a small fraction had a preference for a noiseless, swift visual display instead of a typewriter, since, in addition, nearly everyone thought that a hard copy of notes was essential.

At this stage of the course, a lack of confidence and slight apprehension of the unknown was apparent. Three quarters of the students thought that being taught in a definite sequence of topics rather than having a choice of their own was preferable and just over half the total suggested that two people seated at one terminal would be a better idea than individual instruction.

3.3.7.2 Post-course attitude. Having been subjected to CAI for three afternoons, the students were able to form a preliminary appraisal of the method of instruction. On the statement that the method was too impersonal, both groups were split down the middle. However, over half of the total agreed that this would not be the case if CAI sessions were reinforced with small tutorial classes.

As for the structure of the CAI course, only 15% of the students indicated a preference for multiple choice questions as opposed to questions where they were asked to construct their own response. For feedback messages, two-thirds preferred something more than just an impersonal "yes" or "no". One out of this majority suggested 'if (the response is) correct, just pass on; if incorrect, (give) an explanation'.

Group C students were split on whether they preferred the structure of the lessons, which was predetermined, or the structure of the examples sessions, where learner control techniques allowed them some degree of freedom. However, in the examples classes, they all agreed that the idea of requesting the help they wanted, not what the machine thought they wanted, was most desirable, although, in defence of the infrequency of use of the help facility, one student added 'I suggest that the word "help" be changed to "advice" and we may have used it more'.

3.3.7.3 Change of attitude over the course. The most striking change of attitude encountered was certainly that concerning the preference of lectures to other methods of teaching. Out of the group B students,

four originally preferred lectures but afterwards only one was of the same opinion. In group C, all showed preference for lectures at first, but half of these changed their mind after the course. The reason for this change of heart seems almost certainly to be that the attractive features of CAI, such as self-pacing, individualised learning, etc., have impressed themselves on the students, although one wonders how much the novelty effect has had on both their performance and attitude.

The students' views on the relative effectiveness of lessons and practical classes provided a further comparison between conventional problems classes and those on a terminal. Out of the six students in group B, four originally estimated that they learned more from examples classes than lessons but two of these changed their mind while one of the others changed the other way. This result is about what was expected, as the examples class structure was already well known to them. In group C, all the students initially considered examples classes more fruitful but after the course two-thirds had had a change of opinion. This may possibly be explained by the fact that by and large group C did not have enough time to complete their sets of problems each day. Also, unlike conventional examples classes with a demonstrator, they suffered from not being able to ask specific questions, perhaps about theory, rather than just access predetermined help messages.

There was quite a shift of opinion on whether CAI courses would be enhanced if two students sat at one terminal. Originally, seven out of the thirteen students thought this way but only one of these did not change his attitude. Another student experienced a change of opinion in the reverse direction. This tends to suggest that the experience of a CAI course is not so frightening as at first thought. Perhaps it was realised that the questions during the lessons were not too difficult and adequate reinforcement was given on incorrect responses.

On the question of whether it is preferable to be taught in a definite sequence of topics rather than have a choice, which group C had the opportunity to compare in the light of their experience, five out of seven originally chose definite sequences but two of these changed their preference after the course. This shift in opinion may be explained by the students' preference for the freedom that learner control techniques offer.

Some interesting changes of attitude were recorded with respect to the interface of student and machine, the typewriter terminal.

All of group B originally reckoned that their typing would be too slow but two-thirds of them changed this opinion. All but one of group C had similar misgivings beforehand but half were converted during the course. The reason for this change was probably that during the course typing was at a minimum, despite the fact that there were not many multiple choice questions included. Quite a few responses required single-value constructed replies containing only a few characters.

As for the irritation that listening to a typewriter for an hour or so at a time might cause, only three out of the total of thirteen thought it would bother them, but afterwards they changed their minds completely.

Discussion of the nuisance in having to wait for the course material to be typed out brought differing overall responses from the two groups. Originally, group B all thought that it would not cause them annoyance but two-thirds of them changed their opinion afterwards. With group C, only one student expressed concern at first. After the course, there was no change in opinion at all. Two reasons spring to mind for this state of affairs. As the operating system tends to improve after 1630 or so, group C tended to have better performance from the machine on the average. Also for some part of their course, the problems classes, the environment

is somewhat different. They are not waiting to see what happens next, but only request as much as they want and when they want it. This learner control aspect may be very important when trying to assess student attitude.

The inadequacies of the terminal typewriter seem to be amply compensated for by the fact that printed notes are generated. Only two out of the thirteen stated that they originally had a preference for a noiseless, swift, visual display even though it would not give a hard copy but even these students were converted during the course. As one student remarked, 'Hard copy is essential'.

3.3.7.4 The structure of the course and the performance of the system. Much information was gathered as to the possible structure and content of future courses.

Of the thirteen students who received CAI lessons, only one thought that the notes he had received were poor. However, the students' opinions on the subject content, as far as volume is concerned, varied widely. An equal proportion, almost, suggested that there was too much, insufficient or about the right amount of subject matter. As for the number of worked examples in the lessons, slightly over half the students considered that there were about the right number, but one thought that there were too many.

Only one student considered that there were too few questions asked during the lessons; the rest were quite satisfied. So much so that everyone thought that the level of difficulty of these questions was about right. However, about two-thirds of the total suggested that in some instances they just did not gather what was required of them.

Slightly over half the students considered that feedback messages appeared with the desired regularity, but, of the others, there was an even split with opposing views. As for the receipt of overall scores on sequences of questions, again slightly more than half were satisfied that this information was given about the right number of times but, of the remainder, the majority would have preferred more performance data.

Only one student considered that the "pauses" did not allow him to proceed at his own pace but eight thought that the number of "pauses" was too large. This may be attributed to the fact that all the students were of the opinion that the response from the system was too slow. In fact, one student suggested that fewer "pauses" would help overcome this particular difficulty.

As was quite apparent from the response files, everyone suggested that more time each day would have helped greatly, although three students did not believe that three days was too short for such a course.

The majority of group C students considered four questions in an examples session was too many but the rest thought this was the right number.

3.3.8 System performance during the course.

The following tables represent to what extent the PIL course used the computer during the three days of instruction.

The first table compares usage between the total number of terminal users in the day, that is from 1000 to 1200 and from 1400 to 2000, and groups B and C, with respect to average terminal elapsed time, average CPU time, average virtual memory integral (VMI) in the CPU state and average virtual memory integral in the wait state. By and large, group B used the terminals from 1400 to 1545 and group C, together with the occasional group B student for part of the time, from 1545 to 1830. However, in this table, there is no confusion between entries for group B and for group C.

The second table compares the average total number of virtual pages on the drum with the average number used for the course, taken at quarter hourly intervals. The two periods of comparison represent those times just specified, for group B alone and for mainly group C with one or two group B members. To obtain a rough estimate of the number of students from the course using the terminals over these periods it may be noted

that if only group B students used the machine about 210 virtual pages would be needed whereas if only group C students were considered this would be about 300 virtual pages. During the course of instruction there were seven students in each group.

	sessions	number of sessions	av. elapsed time (min)	av. CPU time (min)	av. VMI CPU (page-sec)	av. VMI wait (page-sec)
day 2	all	266	31	0.55	981	43907
	group B	10	62	0.47	819	107400
	group C	7	174	1.22	2859	405286
day 3	all	281	32	0.51	974	45601
	group B	11	69	0.50	877	121545
	group C	7	158	1.18	3275	449273
day 4	all	281	32	0.51	920	44617
	group B	8	89	0.71	1247	157002
	group C	7	160	1.34	3390	417857

	time	av. total virtual memory (pages)	av. virtual memory for course (pages)	ratio (%)
day 2	1400-1545	676	187	28
	1545-1830	696	278	40
day 3	1400-1545	595	209	35
	1545-1830	749	344	46
day 4	1400-1545	700	208	30
	1545-1830	679	287	42

The most interesting observation is that, whereas the average elapsed time of a group B session is about twice that over all sessions, the CPU time used is of the same magnitude. Similarly, average elapsed time of a group C session is about five times that over all sessions but CPU time used is only twice as much. This confirms that CAI programs require a much smaller ratio of CPU time to elapsed time then the "average" job. The reason is that such programs are concerned mainly with input/output and only need a small amount of computing for response matching and conditional branching from time to time.

In contrast to the small amount of CPU time needed, the students on the PIL course were each required to have a considerable amount of virtual memory. In the case of group B, each student had a constant 30 virtual pages throughout the course. A group C student needed 40 when he started, then up to 50 by the end of the course. This is explained by the fact

that the PIL interpreter was available to him in addition, and use of successive PIL files to store the examples necessitated more and more work space for the interpreter. The figures given for the VMI in the CPU state and the VMI in the wait state provide interesting comparisons. For group B, whose sessions are twice the length of the average session but only use the same amount of CPU time, the VMI in the CPU state is of the same order but the VMI in the wait state is about three times as great. This suggests that even though the same virtual memory is being efficiently used, more is being required just to sit on the drum waiting. For group C, whose sessions are five times the length of the average session but only use twice as much CPU time, the VMI in the CPU state is three times as great as the average but the VMI in the wait state is nine times as great. This suggests that, in proportion, more than the average virtual memory is being used efficiently by members of group C but nearly twice the average over the whole session is required to wait on the drum.

Unfortunately, the number of drum reads is not available at all for comparison. However, it may be that with such large amounts of virtual storage needed, the small amount of CPU time required may be offset by the amount of paging needed. Certainly, the introduction of shared code, which would vastly reduce the amount of virtual memory needed per student, would greatly increase the efficiency in the use of the computer for such a course.

The excessive use of virtual storage is shown in table 2 for each day. Between 1400 and 1545, 25% of the terminals, that is, those being used for the course, used 30% of the total virtual memory on the drum whereas between 1545 and 1830, about 30% of the terminals used slightly over 40% of the total virtual memory.

The following table gives the cost of computer facilities used to run the course. The rates used are those for University departments, these being approximately one third of the industrial rates.

	group B (£)	group C (£)
terminal elapsed time	21	34
CPU time	16	26
VMI, CPU	4	9
VMI, wait	99	240
file storage	2	3
system file storage	6	6
total	148	318

The outstanding feature of this table is the high percentage of the total cost that the VMI in the wait state contributes, yet again suggesting that shared code is essential for future investigations.

The total student hours logged for the course using CAI were 34.7 and 57.4 for groups B and C, respectively. Thus, the corresponding cost in pounds per student hour was 4.3 and 5.4. This includes the charge for hardware, system software and operation and maintainence but not for instructional software. It would be difficult to obtain an estimate for this as not only is it difficult to fit a charge to the number of hours of author time used but also in our case the number of student hours actually used is far smaller than those which might have been used and, indeed, might continue to be used. The National Council for Educational Technology (1968) estimated the cost per terminal hour in a University using ten typewriter terminals as £1.5, excluding cost for instructional software. Thus, the cost for the course was far in excess of their estimate. However, three points of difference should be stressed. Firstly, their estimates were for a dedicated system as opposed to a general-purpose time-sharing system. Secondly, introduction of shared code would reduce the cost by at least half the total (that is, the cost of the VMI in the wait state with shared code would only be about a quarter of its previous value). Finally, two years have elapsed since the estimate was made.

As a contrast, the cost of conventional teaching in the sciences at a University NCET estimated to be £0.7 per student hour.

3.4 Conclusions from the investigation.

One important function of the investigation was an evaluation of the NUTS author language and system in general.

The production of five hours of CAI lessons, which totalled 3300 source statements, and about four hours of on-line example classes, another 1500 source statements, took about 600 hours in all. This included

(i) preparation of subject material, which was already well known to the author;

(ii) coding in the author language, which should have been made easier by the fact that the author was also the designer;

(iii) punching, which was carried out by experienced key-punch operators;

(iv) debugging;

(v) testing, by numerous people, and

(vi) recoding, where necessary.

Thus, taking all these facts into consideration, a slightly larger production time would normally be the case. As it was, a ratio of about 70-1 for the production time against the course time was achieved.

The scope that the author language gave enabled a wide variety of question types to be attempted. Response checking was coped with quite adequately and the only two questions for which correct answers were not credited during the course were amended quite easily afterwards.

In general, the response time given by the time-sharing system was too slow. This certainly can not be accounted for on the basis that more load was being put on the system as the ratio of the CPU time to elapsed time was of the order of 120 to 1 over all the students. However, a heavier load was put on the drum but a measure of this load was not available. Shared code among the participating students would alleviate the problem to some extent, but time-sharing systems usually suffer from a misuse of the facilities by terminal users in general.

Whilst the course was being debugged and tested, it was very difficult to estimate with any degree of accuracy the length of the constituent lessons. Even though a number of different people went through the course, there were at most two of them on terminals at any one time. Comparing this situation with the course proper when at least seven students were receiving instruction at any one time, it can easily be seen that a false estimate of the length was obtained.

In an effort to improve the service given by the time-sharing system and, hence, overcome the shortage of time, certain organisational changes were successfully made. The chief of these was that of allowing the students to commence promptly when the system was ready, before the general users were permitted to begin. This greatly reduced competition for the loader. Another move was to advise general users with large virtual memories to terminate their sessions and return later if they needed a terminal.

The general conclusion on the use of NUTS is that it may easily be used to generate instructional programs. As to the performance of the dialogue, though this is not as good as desired, the causes have been mentioned above and a future investigation would certainly include their improvement.

The second important purpose of the investigation was to compare different methods of teaching programming languages. Such features as CAI lessons and the on-line availability of the language processor and a "help" facility were used together with the control methods of conventional lectures and examples classes with demonstrators.

An analysis of covariance revealed that there was a significant difference between the performance scores of the conventional class and those for the CAI groups. A positive response was achieved by the control group. This is what was expected, as lectures do not make any

extra concessions to the poorer students and, normally, good students do well no matter what teaching method is used. The students from the CAI groups gave a negative response. This suggests that the poorer students benefitted greatly from the individualised instruction they received. They chose their own routes by virtue of the different combinations of right and wrong answers and continued at their own pace as necessary. The puzzling feature of the results from the two CAI groups is that the students with higher aptitude showed up poorer than expected. The most likely reason for this is that the course failed to give these students the motivation to do well throughout the week, and hence they became less interested. This theory is borne out by certain results taken from the post-questionnaire of those five students who fall into that category which had in it students of above average aptitude but below average performance. The co-ordinates of these students, as described in 3.3.3, are (64,40), (77,43), and (74,57) from group B and (86,51) and (80,57) from group C.

On the question of whether it is a nuisance having to wait for the typewriter to produce the notes, four out of five agreed with this opinion but, of the remaining eight, as many as seven disagreed. Similarly, four out of the five suggested that their typing was too slow for CAI techniques yet seven of the remainder were against this suggestion. These two opinions, coupled with the fact that the split was four to one for the view that the method was too impersonal yet three to five against this from the rest, do certainly point to the suggestion that the attitude after the course of these five was one of disinterest and, perhaps, boredom.

In the light of this, it seems that the question of motivation should be studied more deeply before any future investigations of this kind are made, but the results to hand tend to suggest that placement of those students whose aptitude scores for programming are poorer than average should be into groups where methods of CAI are being employed

so that they may benefit from the individualised, self-paced instruction. The rest may be given conventional lectures and demonstration classes without detriment.

Other information returned from the response files suggests that:

(i) performance score did not seem to depend on the amount of the course that the student had been able to get through;

(ii) there may be a dependence of performance score on the percentage of questions the student answered correctly at the first attempt;

(iii) performance score may be inversely proportional to the average time to make a first attempt;

and (iv) there is a significant negative correlation between performance score and average total response time per question.

These relationships, together with the fitted lines from the analysis of covariance, may possibly be used to suggest how well the students are doing during the course and especially may be used if a student is absent for the final performance test. For example, for the student from group B who failed to appear for the post-test, 95% confidence limits for her score from each of the above relationships are:

(i) analysis of covariance, group B slope: (56.17,70.75);

(ii) percentage of questions correct at the first attempt: (54.49, 71.09);

(iii) reciprocal of the average time for the first attempt: (56.07, 72.81), and

(iv) average total response time per question answered: (60.29, 79.61).

The study of the students' attitudes and their change in attitude over the course brought some interesting results. Most striking was the change in attitude of preference of one type of teaching to another, in our case, from lectures to CAI methods. Certainly, the majority of students seemed to take to CAI and its advantages but, of course, it is impossible

to eliminate novelty effects from this one investigation. As for examples classes, the students who had the conventional sessions were virtually unchanged in their estimate of relative usefulness of these as compared with lectures, but the students who had problems sessions using CAI shifted completely from their original estimate that more is usually learned from examples classes than from lectures. However, these latter students did agree that such a "help" facility as was available, one in which the student had control, was most useful. The main reason for this change in opinion is probably that the CAI examples classes lacked the feature of the student having free, unlimited access to a two-way conversation, as given by a demonstrator.

On the question of whether CAI is too impersonal, there was an even split in both groups but over three-quarters of the students felt that, interspersed with small tutorial classes, CAI courses would not be too impersonal at all. This suggests that any future investigation should not attempt to provide instruction without adequate human interaction during the course and, in fact, reinforces the theory that CAI should never be allowed to replace conventional teaching but only supplement it, paying particular attention to those areas where conventional teaching is not so effective.

The main impression received regarding the interface between student and machine was one of reasonable satisfaction. Almost two-thirds of the total were not bothered about having to wait for the subject material to be typed out and the same number thought that their typing was good enough. No one was disturbed by the noise of the typewriter. In fact, everyone showed preference for a terminal typewriter rather than a visual display, the main reason being that hard copy of notes is considered essential.

Of the course in general, the most obvious conclusion is that more time would have been a great advantage but this restriction was imposed by

the non-availability of the students for any longer than five afternoons. The situation would have been eased by a shorter course, but an exact estimate of its length was not available from the pre-course trials.

The extent to which the group C students used the "help" facility was rather disappointing. Admittedly, 41% of all examples attempted were answered correctly with no help at all, and a further 18% were solved after INFO had been used, but it is the remaining 41% that causes concern. Although another 18% were solved after use of the "help" facility, still 23% remain. Perhaps the solution to the problem does lie in the suggestion made by a student that the "help" facility should be renamed the "advice" facility, but there is always the basic difficulty that students prefer to be asked if they require assistance; they do not like to ask themselves. Obviously, the request mechanism as provided does not alleviate this psychological barrier.

CHAPTER 4. From author languages to easy author entry systems.

4.1 Introduction.

To be effective in computer-assisted instruction and programmed instruction, an author requires a rather severe self-discipline and a considerable amount of specialised knowledge. He must first of all know his subject matter and be a good writer. Beyond these basic requirements, a programmed instruction writer must know how to write to stated objectives and how to ask meaningful questions. In addition to this, a CAI author must know how a computer operates and the specific details of the computer language he is using. It is unreasonable, perhaps, to ask an author to possess so many skills in order to begin to write for CAI. Indeed, few authors who are not already experienced computer programmers have been persuaded to take the time and effort to learn a CAI language. The result has been that CAI materials have been on the whole written by people conversant with computers rather than by good teachers conversant with their disciplines.

To overcome this problem to a certain extent, the author who does not know a CAI language has usually seconded assistance from a number of different people to get his subject content from the initial draft into the computer memory successfully.

Firstly, the author must have an editor to perform a grammatical edit of his subject matter. The corrected draft must then be typed and given to a programmer who converts the English statements into valid computer program statements. If not coded on-line, the program must be punched into cards, entered into the computer by an operator, and checked for validity in the programming language. If errors exist, someone must debug the program, make the necessary changes and re-test. Finally, the material is returned to the author who checks it for inconsistencies in content and logic before it is released to the students.

For the author who has not been persuaded to learn an instructional coding language, a new system has been devised which precludes the need for considerable assistance as outlined above. The system allows the generated material to make effective use of the power of a CAI system, which, in this instance, is COURSEWRITER II for the IBM 1500 System. However, the problem and proposed solutions are not restricted. The devised system also dispenses with the necessity of having a programmer produce COURSE-WRITER II statements and then someone to debug the program for language errors. This is simply achieved by providing the author with a Course Planning Form on which he may enter his subject presentation, his questions and expected answers, and the corresponding courses of action in almost unlimited format.

4.2 Previous easy author entry systems.

Not a great deal of research has been carried out in the past to provide easy author entry.

Perhaps the easiest system to use from an author's point of view is that designed by Dean (1969). Authors fill in planning guides, on each page of which they indicate some form of identification, text presentation, anticipated responses and resultant branching. These are then edited for grammar, syntax and spelling before a cardpunch operator converts them by punching one card for identification and one card for each line of text on the planning guide. An editor uses standard cards from a pre-punched supply to make the author's deck ready for assembly. The immediate advantages of this system are that any card-punch operator can make up about 90% of the required cards directly from the planning guide, and an editor can supply most of the rest without coding. The small amount of coding required is nearly automatic since it consists of entries taken in sequence directly from the author's manuscript. The disadvantages are that the code generated is not very elegant, consisting of macros to a large extent, and that there is not enough variety offered for response matching. Only exact keyword and lightpen responses may be specified.

Other easy author entry systems have been by and large more general CAI systems but with a distinct emphasis on ease of code generation. They include those designed by Kerr et al. (1969) and Meadow et al. (1968). Descriptions of both of these appear in section 2.1.4.

4.3 The Course Planning Form.

Up to the present, most authors have prepared course material in one particular format, namely, the presentation of single frames to the student. Then, if he answers successfully, the student is allowed to continue to the next frame in sequence. If he does not, he sees a frame containing remedial information. As this format for course preparation is in common use, the system to be described provides a process which will quickly generate such course materials. The objective is that any author should be able to prepare CAI course material in a form which, when punched into cards, is immediately converted to COURSEWRITER II by this system. Such a system is properly termed a pre-processor.

The layout of the Course Planning Form was determined by the technical specifications of the 1510 CRT Display Unit. The 1510 is divided into 32 rows and 40 columns of addressable spaces. However, as two rows are required per character, the 1510 effectively displays 16 rows by 40 columns. This determined the layout of the planning form inasmuch as one space is provided for each character, but half-line shifting for superscripting or subscripting is still available despite the fact that spaces do not exist for this purpose on the form.

As suggested by Dean (1969), far from imposing a restriction on the author by insisting that he works within the confines of a 16 by 40 character form, the discipline imposed by the form may well assist him in his efforts to communicate with students. Since the student can see but one frame at a time, it is important that the author provide a clear, meaningful presentation in each frame. If he is too verbose, he will be unable to complete his presentation in sufficient space to appear as a single display. The 16 by 40 form serves to remind him when such an event will occur.

The form is divided into four sections.

(i) Identification: this section must be completed.

(ii) Presentation: this section must include entries if any instructional material is to be shown.

(iii) Decision: the entries tell the computer which frame to display next.

(iv) Response analysis: the author specifies his contingency prescriptions:

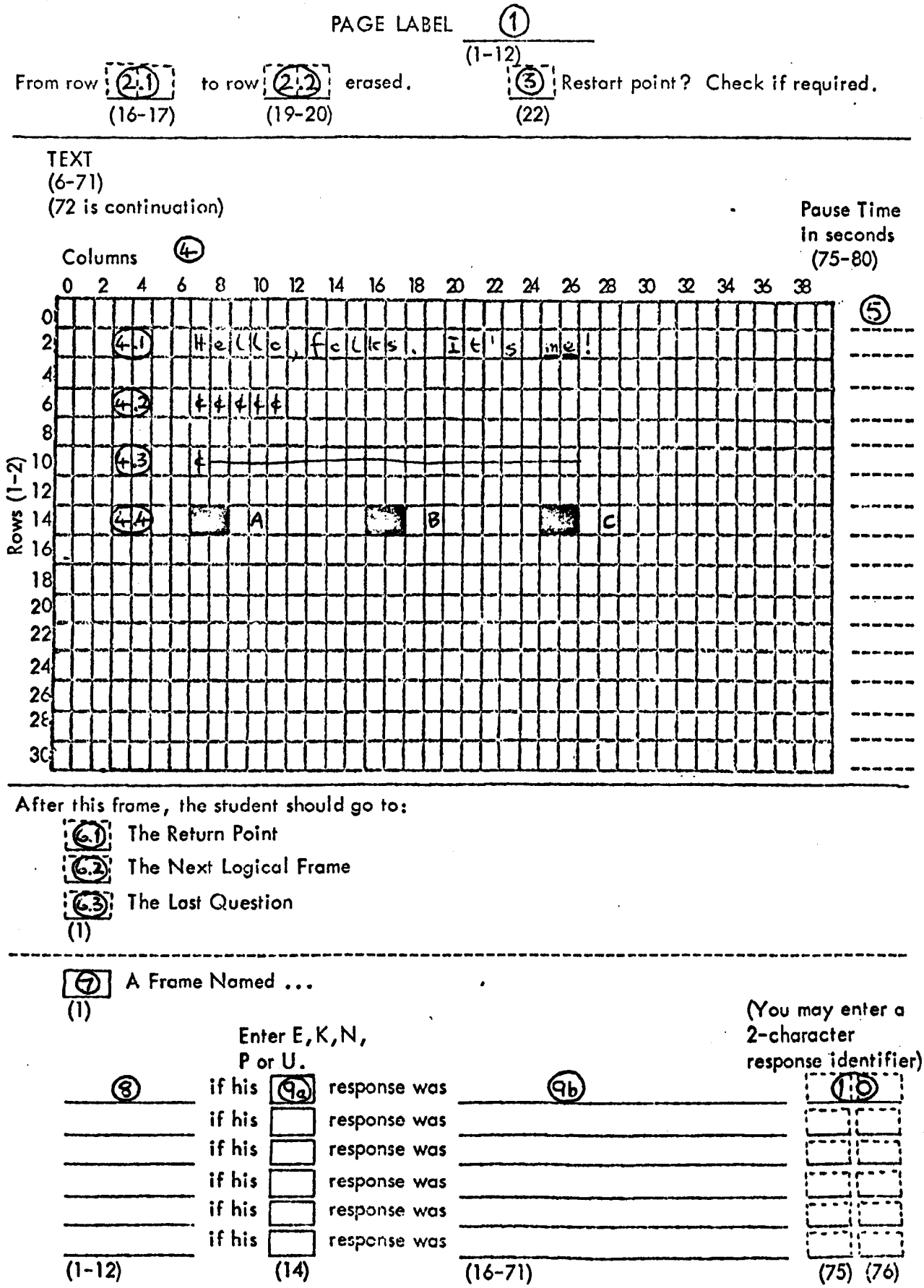
Sections (iii) and (iv) are alternatives. Either section may be used but not both.

In an effort to make the form easier to use, mandatory parts have been assigned solid boxes or underlining, whereas optional parts are indicated by broken lines. A Course Planning Form is shown in Figure 4.1.

The numbers contained in parentheses under each entry indicate the particular card columns the keypunch operator must use to record that information. (An optical scanner would allow the keypunch operator to be bypassed completely). Corresponding to the four sections of the form, there are four types of card produced: one only for section (i) of the form; any number (including zero) for section (ii); one only for section (iii); and at least two cards for section (iv), one of which is similar to that for section (iii) plus at least one for the response analysis.

Naturally, it is expected that the form, being of a general nature, will not allow the author to create all types of teaching procedures. However, once the author has become proficient in the instructional coding language, he can then use hand coding techniques on the code produced from the form via the pre-processor.

Figure 4.1


PAGE LABEL (1)
(1-12)
From row (2.1) to row (2.2) erased.
(16-17) (19-20)
(3) Restart point? Check if required.
(22)
TEXT
(6-71)
(72 is continuation)
Pause Time
in seconds
(75-80)
Columns (4)
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
Rows (1-2)
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
(4.1) Hello, folks. It's me!
(4.2)
(4.3)
(4.4) A B C
(5)
After this frame, the student should go to:
(6.1) The Return Point
(6.2) The Next Logical Frame
(6.3) The Last Question
(1)
(7) A Frame Named ...
(1)
(You may enter a 2-character response identifier)
(8) if his (9a) response was (9b) (10)
if his response was
if his response was
if his response was
if his response was
if his response was
(1-12) (14) (16-71) (75) (76)

4.4 A guide for authors.

The following explanatory notes correspond to circled numbers on the Course Planning Form in Figure 4.1 (The numbers are used in this instance merely to facilitate the following discussion).

1. The page label must appear on every sheet except a continuation sheet (see note 9. condition (v)). The label is automatically displayed in columns 34-39 of row 0 to aid debugging. The display format is as follows. The page label is an unsigned integer optionally followed by a string of alphabetic characters (either upper or lower case), the total length being not more than six characters. The purpose of insisting on unsigned integers as labels is to provide notation in ascending order of magnitude, which is the logical sequence of the course as written by the author. The first page label in a particular frame will be the unsigned integer itself, but any subsequent pages may be labelled by the unsigned integer followed by a string of letters. This is to allow the author freedom to use learner control techniques as suggested by Grubb (1968). Among other things, the student will be able either to branch backward to the last logical frame or to skip forward to the next logical frame.

2. Before presentation of any text, the author specifies which rows (if any) he wants erased.

   2.1 Here he enters the first row of the sequence. It is a number greater than or equal to zero but less than or equal to 31. If there is no entry, a default value of zero is assumed.

2.2 Here he enters the last row of the sequence, also a number greater than or equal to zero but less than or equal to 31. Naturally, the number entered in 2.1 must be less than or equal to that entered in 2.2. If no entries appear, all 32 rows are erased. Both entries must be zero to produce no erasure at all.

3. If the author desires this point in the course to be a restart point, he ticks the box. At a restart point all current information about the student and the course is stored in the 1500's student record file. Consequently, if he stops, whether through choice or system failure, the system will restart him at this point, with all up-to-date information, when he decides to return.

4. The author fills in the form in exactly the same way he wishes to present his material. He must not forget that each character requires two rows on the screen, but to aid him the form is divided into double rows as is indicated by the left margin row numbers.

4.1 He writes small letters, capital letters, punctuation, underlining, subscription, superscription, etc., as required.

4.2 To denote where the cursor (moving indicator) is to appear on the screen and the subsequent answer space for a keyboard response, the author fills in the appropriate positions with a "¢".

4.3 A shorthand form of this is to mark the cursor position with a "¢" and then draw a line throughout the rest of the required answer space.

4.4 To denote a light patch area for a light pen response, the author shades in the appropriate positions, as shown.

Of course there are some necessary restrictions.

(i) The author may not specify both a keyboard and a lightpen response simultaneously.

(ii) For a keyboard response, there may be only one continuous answer space. In other words, no embedded characters or blanks are allowed. However, responses larger than one line are permitted.

5. The author has the option to specify pause times, in seconds, for each line of the display. The pause provided by the pre-processor after the last line continues until the student presses the space bar to continue.

6. The author should complete either the Decision section or the Response Analysis section, not both. If there is no response required of the student, then the author must fill in the Decision section by ticking one, and only one, of the branching boxes.

6.1 The Return Point box is ticked in situations such as the following. An author may wish to generate the same comments whenever he receives a certain response. To save repetitive generation of the same coding, he may initially jump to a frame where the comments are generated and from that frame "return" to the next logical frame after the one from which he made the initial jump.

6.2 The Next Logical Frame box is ticked when the flow is to be to the first frame in the next logical section. For example, it may be used when there was no question asked, but merely text presented at label 6. Then the Next Logical Frame is 7. If the author was supplying the correct answer at label 6xy, then the Next Logical Frame is still 7.

6.3 The Last Question box is ticked when the author wishes the student to attempt the last question again. Usually, the current frame will be some remedial hint after an incorrect response was diagnosed.

7. The Response Analysis section is filled in when response processing is required after a question. The Frame Named box is ticked and there must follow at least one entry in the matching specification list.

8. Here, the author enters the page label of the frame that is to follow if the student's response is successfully matched. It follows the same format as the page label entry in 1.

9. To determine the type of response processing required, the author writes into 9a) one of the following letters: E, K, N, P or U; and then places into 9b) the actual "required response" characters.

9.1 E. An exact keyword match is required. The characters, including any required blanks, are written in 9b). If this is left blank, then any string of characters the student enters will provide a match.

9.2 K. This allows for misspelling, etc. A "kernel" match is permitted, but the essential characters being sought (no blanks) are written in 9b). For example, if the author seeks "FORTRAN", then he may enter "FTRN" in the hope that the student may obtain a match even allowing for incorrect characters, missing characters, etc., usually caused by misspelling or typographical errors. So long as the specified characters appear in order somewhere in the student's response, but without embedded blanks, a match is considered to have been made.

9.3 N. This allows a search for a particular numerical value. Two types of checking may be specified. Firstly, if an

exact value is requested, the author specifies the value. A match will occur if the response contains a numerical constant within the interval formed by the value plus or minus half the least significant power of ten given in the value. Secondly, if a range will suffice, the author enters the lower bound and the upper bound separated by a minus sign. This causes a match to occur if the response contains a numerical constant greater than or equal to the lower bound and less than or equal to the upper bound. However, if the author requires one exact value, that is, no possible error, he must use checking of the second type and make the lower and upper bounds both equal to the exact value sought. In addition to the choice of a numerical check, the author may test the truth of the disjunction of some of each or both by separating each value or range by a comma. For example, he may enter:- 273, 31.2-37.6, -21--18, 98.4

9.4 P. This signifies a light pen response. In 9b) the author enters the coordinates in the form "row, column" of any point in the light patch with which he wishes to associate this attempted match.

9.5 U. This is entered to denote an unanticipated response. Here 9b) is left blank.

There are a few conditions imposed on the use of these five kinds of entries.

(i) There is no default for an entry in 9b) except for E and U.

(ii) Lightpen response matching cannot be mixed with any type of keyboard response matching.

(iii) For keyboard response matching, all U's must appear last in sequence; that is, no E, K or N may follow a U.

(iv) For lightpen response matching, only P may be entered in 9a). No U's are allowed. However, the pre-processor produces code which displays on row 30 the suggestion that the student re-answer if he pointed to anywhere except to a specified light patch.

(v) There is no limit to the number of response processing entries there may be. If one form is insufficient, the author may use additional forms.

10. The author may enter a two-character response identifier in columns 75 and 76. Then when the computer-listed student response record is made available, the author can quickly determine how the student answered each of the questions.

4.5 A guide for key-punch operators.

This section demonstrates the ease with which information is transferred from the Course Planning Forms to card input by the keypunch operator.

There are four types of card, each corresponding roughly to each section of the form, if used. Each input deck is ended by a card containing only "*" in column 1.

4.5.1 Identification card.

(i) Page label: columns 1-12; use "<" for upshift, ">" for downshift.

(ii) Starting line for erasure: columns 16-17; for a one-digit number, use either column.

(iii) Finishing line for erasure: columns 19-20; for a one-digit number, use either column.

(iv) Restart point: any character other than blank in column 22. All other columns must be blank.

4.5.2 Presentation card.

The keypunch operator must begin the line at the lowest row number, that is, that part of the line nearest the top of the form, and work down the form if necessary by indexing. This is important, of course, when superscripting has been used. The following conventions apply.

(i) for upshift use "<".

(ii) for downshift use ">".

(iii) for a keyboard response use "¢" but where the author's shorthand notation is used all "¢"s must be entered.

(iv) for lightpen response use "¬V".

(v) for index use "|".

(vi) for reverse index use "!!!!".

(vii) for backspace use "%".

(viii) for multiply sign use "@".

(ix) for divide sign use "<@ " .

The card layout is as follows.

(i) Row number: columns 1-2; for a one-digit number either column may be used. If the keypunch operator does not specify the row number, it will default to the "last one used" +2.

(ii) Text: columns 6-71, using the given notation.

(iii) Continuation: any character in column 72. This will be necessary if many upshifts, downshifts, backspaces, etc., are needed.

(iv) Pause duration: columns 75-80. For less than five-digit numbers, any consecutive columns out of columns 75-80 may be used.

4.5.3 Decision card.

Column 1 is used on this card. All others should be blank.

(i) For "return point" enter "R".

(ii) For "next frame" enter "N".

(iii) For "last question" enter "Q".

(iv) There is also a fourth alternative which is used when the Response Analysis section has been filled in. It is the first card corresponding to that section. Thus, for "a frame named..." enter "T".

4.5.4 Response Analysis card.

(i) Page label: columns 1-12.

(ii) Type of response matching: in column 14 either "E", "K", "N", "P" or "U" is entered.

(iii) Text: columns 16-71.

(iv) Two-character response identifier: columns 75-76.

All other columns must be blank.

4.6 The action of the pre-processor.

The pre-processor was written in FORTRAN because this language

(i) is universal and thus allows ease of communication between programmers,

(ii) is easily and readily debugged,

(iii) allows the addition of other facilities as further developments are tackled in the project, and

(iv) is supported on the IBM 1130 or IBM 1800 CPU used with 1500 Systems.

The current version of the pre-processor comprises approximately 1500 basic FORTRAN IV source statements. The pre-processor's action is as follows.

1. Read in an Identification card. If it is the last card, pass to 12.
2. Check for a valid label name, whether erasure is requested and whether a restart point is required. Check parameters for COURSEWRITER II "de" (display erase) instruction. If no errors, write each card image to disk and give a listing of it on the printer. Otherwise, give appropriate diagnostic message with the card number and, in some cases, the column number. Also increment the error count. In either case, pass to 3.
3. Read in a card. If it is the last card, pass to 12.
4. Check to see whether the card read in is a Decision card. If it is, pass to 6.
5. Check for valid row number (or defaulted row number), response requests, text presentation and pauses. Check parameters for the "dt" (display text) instruction. Either write to disk or give diagnostics. Pass to 3.
6. Check to see whether the Decision section or the Response Analysis part has been filled in. If the Decision section has been completed, generate the appropriate branch instruction and pass to 1.

7. Read in a Response Analysis card. If it is the last card, pass to 12.

8. Check to see whether there is a response type entry. If not, it must be an Identification card, so pass to 2.

9. If this is not the first Response Analysis card of the series, pass to 11.

10. From the information given by the Presentation cards, generate the instructions "dl" (display emphasis line) if the keyboard response is a one-line insertion, and either "ep" (enter and process response) for a keyboard response, "epi" for a keyboard insertion response or "epp" for a lightpen response. Write to disk or give diagnostics as necessary. Immediately before the enter and process response instruction the label name of the next logical label in sequence is loaded into return register 0 so that any future branching back to "return point" will be meaningful. Just after a keyboard enter and process response instruction, a macro call of "ercalc" (see 4.7) is generated.

11. Check for valid label name, see which type of response analysis is requested and generate, if possible, the appropriate instructions.

(i) For "E", "aa" (anticipated answer).

(ii) For "K", a "ld kernel characters" (loads text into buffer) followed by a call of the function "keyl" (keyletter).

(iii) For "N", a call of the function "lt" (limit).

(iv) For "P", "aap" (anticipated answer, lightpen).

(v) For "U", "un" (unrecognisable response).

Before the first "un", a macro call of "emany" (see 4.7) is inserted. After the particular instruction(s) specifying the analysis is generated, the branch-to-label name is formed. Either write to disk or give diagnostics. Pass to 7.

12. Check the error count to see if there have been any errors. If so, terminate, but otherwise produce a punched card deck from the card images resident on disk, and then terminate.

The pre-processor will find only one error per card. When an error is detected, the card is "rejected" and the pre-processor reads in the next card. Naturally, this may produce further errors, but only in the current logical frame. For each error an appropriate diagnostic message is listed with the card number. In some cases, where meaningful, the column number is also given.

4.7 Macros used.

The research being undertaken by the IBM Education Research Department, San Jose, is geared towards the use of learner control techniques. The main effect for the student is to allow him to move freely throughout the course, that is, to skip forward, move backward, proceed to the glossary, return to the course outline map, etc.

In an effort to coordinate the course material produced by the pre-processor with the requirements of the learner control coding, the macro "emany" is called before an unanticipated response is produced. This macro searches the response to see whether the student called upon any of the above mentioned utilities and then performs whatever action is required. The three parameters necessary for "emany" are the "current frame", the "next logical frame" and the "last logical frame". Since the author uses sequential numbering for his logical frames, these parameters are readily available and provided by the pre-processor.

A second facility provided is that in which the student's response, if it contains an arithmetic operator, is fed into an arithmetic syntactic analyser from which the equivalent reduced value is returned to the calling program to be checked in the response analysis. To effect this, a call is made upon the macro "ercalc" immediately after the response is entered. No parameters are required for this macro.

4.8 Sample input and output.

There follows part of a simple example demonstrating the use of the pre-processor. In Figure 4.2 is some of the author's input on Course Planning Forms. Figure 4.3 shows the printout of the keypunch operator's card deck. The corresponding output from the pre-processor is shown in Figure 4.4 with the COURSEWRITER II instructions and also the diagnostic messages.

Figure 4.2

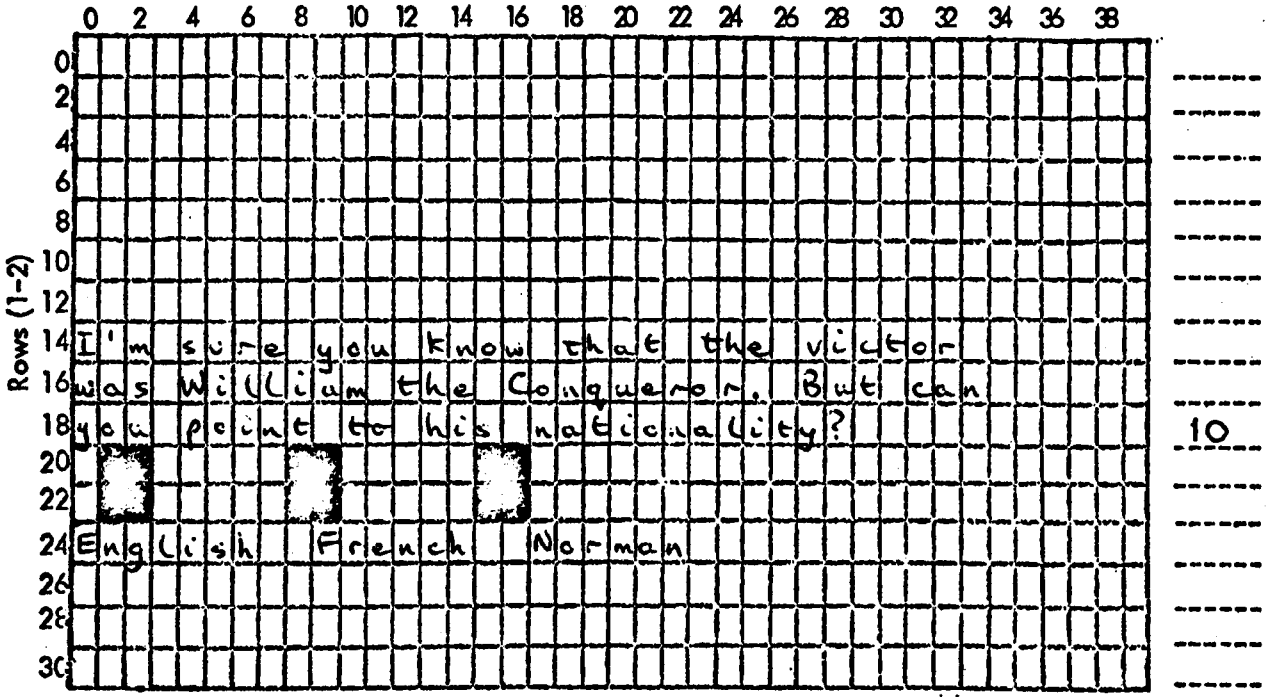
PAGE LABEL 3 (1-12)

From row [ ][0] (16-17) to row [ ][0] (19-20) erased. [✓] (22) Restart point? Check if required.

TEXT
(6-71)
(72 is continuation)

Pause Time
In seconds
(75-80)

Columns



After this frame, the student should go to:

[ ] The Return Point
[ ] The Next Logical Frame
[ ] The Last Question
(1)

[✓] (1) A Frame Named ...

					(You may enter a 2-character response identifier)	
	Enter E,K,N, P or U.					
3a	if his	P	response was	20,16	N	
3b	if his	P	response was	21,9	F	
3c	if his	P	response was	23,32	E	
3c	if his	U	response was			
	if his		response was			
	if his		response was			
(1-12)		(14)		(16-71)	(75)	(76)

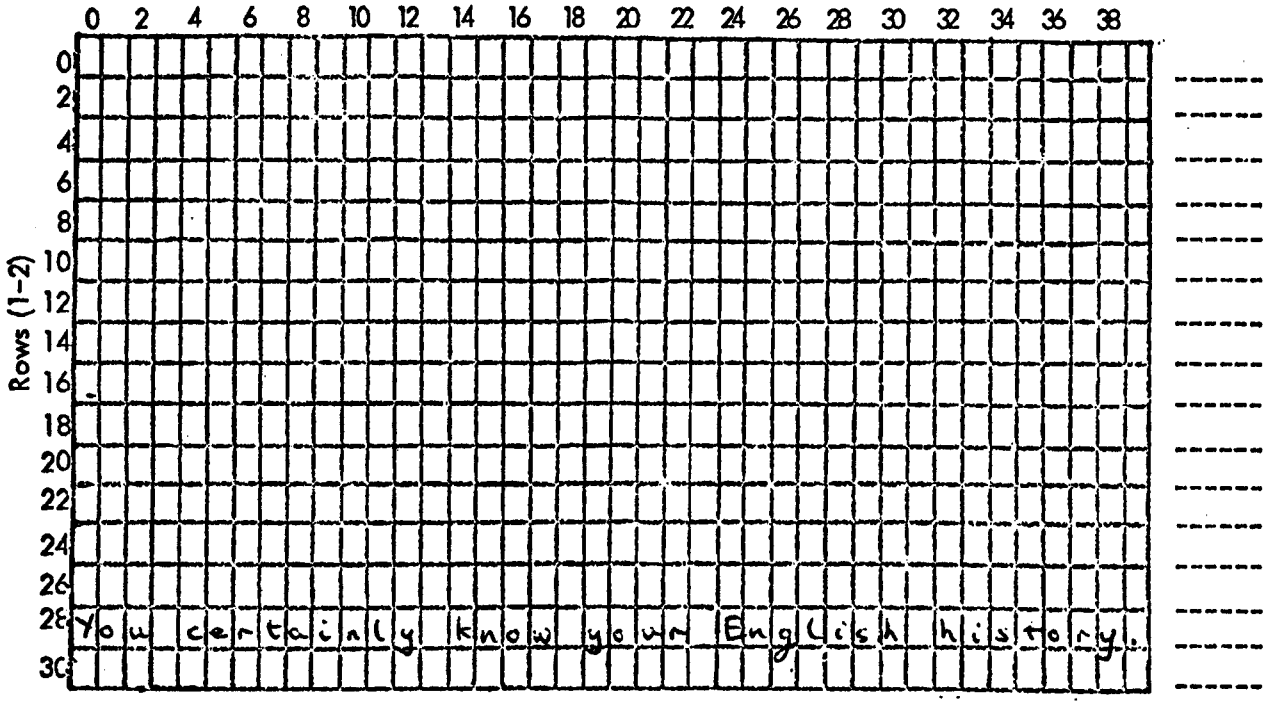
Figure 4.2 (cont.)

PAGE LABEL 3a (1-12)

From row [0] (16-17) to row [0] (19-20) erased. [ ] (22) Restart point? Check if required.

TEXT
(6-71)
(72 is continuation)

Pause Time
In seconds
(75-80)



After this frame, the student should go to:

[ ] The Return Point
[✓] The Next Logical Frame
[ ] The Last Question
(1)

[ ] (1) A Frame Named ...

					(You may enter a 2-character response identifier)
_____	if his	[ ]	response was	_____	[ ][ ]
_____	if his	[ ]	response was	_____	[ ][ ]
_____	if his	[ ]	response was	_____	[ ][ ]
_____	if his	[ ]	response was	_____	[ ][ ]
_____	if his	[ ]	response was	_____	[ ][ ]
_____	if his	[ ]	response was	_____	[ ][ ]
(1-12)		(14)		(16-71)	(75) (76)

Keypunch operator's card deck

```
3                  0  0 R
14   <I>'M SURE YOU MUST KNOW THAT THE VICTOR
     WAS <W>ILLIAM THE <C>ONQUEROR. <B>UT CAN
     YOU POINT TO HIS NATIONALITY?                                    10
      ¬V¬V      ¬V¬V      ¬V¬V
      ¬V¬V      ¬V¬V      ¬V¬V
     <E>NGLISH  <F>RENCH  <N>ORMAN
T
3A              P 20,16                                               N
3B              P 21,9                                                F
3C              P 23,32 ←CARD 53                                      E
3C              U       ←CARD 54
3A                 0  0
28   <Y>OU CERTAINLY KNOW YOUR <E>NGLISH HISTORY.
N
3B                 0  0
28   <V>ERY NEARLY  <H>E CAME FROM <F>RANCE BUT
     WASN'T EXACTLY <F>RENCH. <P>LEASE TRY AGAIN.
Q ←————————————— CARD 62, COLUMN 2
3C                 0  0
28   <G>OOD HEAVENS. <P>ERHAPS YOU HAD BETTER
30   READ UP ON THIS.
```

Figure 4.3

COURSEWRITER II instructions and diagnostic messages

```
.
.
3
        PRR
        DT  0,34¬/¬/¬/3
        DTI 14,0¬/2,14¬/38,0¬/<I>'M SURE YOU MUST KNOW THAT THE VICTOR
        DTI 16,0¬/2,16¬/34,0¬/WAS <W>ILLIAM THE <C>ONQUEROR. <B>UT CAN
        DTI 18,0¬/2,18¬/29,0¬/YOU POINT TO HIS NATIONALITY?
        PA  100
        DTI 20,1¬/2,20¬/16,1¬/¬V¬V      ¬V¬V      ¬V¬V
        DTI 22,1¬/2,22¬/16,1¬/¬V¬V      ¬V¬V      ¬V¬V
        DTI 24,0¬/2,24¬/23,0¬/<E>NGLISH  <F>RENCH  <N>ORMAN
        PAE
        LR  4¬/RRO
        EPP 9999¬/3
        AAP 4,20,2,15¬/N
        BR  3A
        AAP 4,20,2,8¬/F
        BR  3B
RESPONSE REQUEST INCONSISTENT WITH RESPONSE INDICATIONS
CARD NUMBER   53
ONLY 'P' SHOULD BE ENTERED WHEN LIGHT PATCH HAS BEEN INDICATED
CARD NUMBER   54
3A
        DT  0,34¬/¬/¬/3A
        DTI 28,0¬/2,28¬/40,0¬/<Y>OU CERTAINLY KNOW YOUR <E>NGLISH HISTORYC
.
        PAE
        BR  4
3B
        DT  0,34¬/¬/¬/3B
        DTI 28,0¬/2,28¬/36,0¬/<V>ERY NEARLY  <H>E CAME FROM <F>RANCE BUT
        DTI 30,0¬/2,30¬/40,0¬/WASN'T EXACTLY <F>RENCH. <P>LEASE TRY AGAINC
.
        PAE
        BR  RE
EMBEDDED BLANKS NOT ALLOWED IN LABELS
CARD NUMBER   62 COLUMN NUMBER  2
        DT  0,34¬/¬/¬/3B
        DTI 28,0¬/2,28¬/36,0¬/<G>OOD HEAVENS. <P>ERHAPS YOU HAD BETTER
        DTI 30,0¬/2,30¬/16,0¬/READ UP ON THIS.
        EN


YOU ENTERED   64 CARDS.
  116 COURSEWRITER II STATEMENTS HAVE BEEN PRODUCED
AND YOU HAD    5 ERRORS.
CARD OUTPUT HAS BEEN SUPRESSED.
```

Figure 4.4

4.9 Conclusion.

This system is a prototype of many other possible systems. It was designed to free the author from dependency on a specific CAI language or any particular computer. This achieves a level of standardisation that is not at present possible with the many variants of COURSEWRITER currently available. Furthermore, the system is self-documenting. This permits reviews to evaluate the program without going through an entire course as a student would or reading a complete computer listing for such purposes.

The current implementation generates COURSEWRITER II for the IBM 1500 Instructional System, but this is the result of a particular implementation and in no sense reflects a limitation on the system. In generic terms, the system is a pre-processor.

A system description similar to that given in this chapter appears elsewhere (Dowsey, 1970b) and an extended version called COURSEMAKER, which is being developed by the IBM Education Research Department, San Jose, is described by Dean (1970a,b,c).

SUMMARY AND CONCLUSIONS.

During the course of this study a teaching system evolved. Information gathered about other systems and a comparison of commands available shows that this teaching system offers as wide a range of facilities as may reasonably be required to use CAI. Authors may design, edit and test CAI lessons in an author language, monitor students' progress through courses of lessons and perform virtually any calculations necessary through the provision of a calculating language, desk machine and the programming language, PIL. Further improvements would not seem to be in the inclusion of more commands but mainly in increasing efficiency. This is largely dependent on the operating system and extension of its facilities. However, future work might include further evaluation studies and more widespread use.

The most important part of any teaching system is its author language. The study of previous author languages pointed out such glaring omissions as a powerful calculational capability, extensive response processing and a facility to allow branching to depend upon some aspect of the student's response history. Perhaps the most significant single addition to this author language is a flexible, comprehensive recording file, the response file. Far too many previous languages have not provided the author with sufficient feedback information, especially that which can be used actually in the CAI program. However, this author language is easy to use, quite readable and is capable of providing any type of dialogue normally associated with CAI. Probably the only improvement in the actual language would be the addition of string manipulation. The choice of FORTRAN for the implementation language proved most satisfactory since the processors were written and debugged in a very short time. This was preferable to gaining a small

amount of time by using a machine code, which would have made the implementation of such features as response processing and text output easier, but losing a greater amount overall in that the whole operation of writing and testing would have been far longer. As efficiency of the processors was not studied in any great detail, since working versions were required as quickly as possible, studies involving such aspects as efficiency in translation and re-translation might be carried out. Probable future work might include an interpretive version of the language or a version in which the intermediate code is stored in direct access fashion so that complete re-translation would not be necessary for correction of just a few errors.

The author language approach to CAI means providing statements, the effects of which the author must understand fully before he can use them to advantage. On the other hand, the easy author entry approach means that the author may write courses much more quickly and easily but there is a trade-off of facilities for the decreased author time and involvement in programming. After a decade of CAI, in which authors in general have not been persuaded to learn author languages, the swing seems to be toward easy author entry systems. Such a prototype system has been provided. It is extremely easy to use, with a simple planning form, and contains about the same response processing facilities as the author language but, of course, not nearly as much decision branching capability. As the uppermost design criterion was to keep it easy from the author's point of view, a degradation in the facilities available was inevitable. Since the system takes the form of a pre-processor, there must be a suitable language which is generated. The language used was not ideal owing to its lack of any kind of performance recording but adequate for a prototype version. The NUTS author language would certainly have been

better as it provides a greater variety of response matching, a powerful performance recording facility and calculational capability. However, COURSEWRITER II does provide for the use of displays and was used because the operating system available was the IBM 1500 Instructional System. There is limitless opportunity for improvements, in providing more facilities, yet still keeping the system uncomplicated. Performance recording would allow more extensive decision branching. Future systems might dispose with the pre-processor idea and produce an object code directly from the form. This could either be a simple intermediate code or an actual machine code.

In the investigation comparing the conventional teaching of a programming language with that using two CAI methods, students with below average aptitude seemed to benefit more from CAI in general but the rest seemed as well off with conventional teaching. Also, there seemed to be little or no difference between the use of on-line examples classes and demonstration sessions. The "help" facility, which was available to one CAI group, was not particularly well used. Perhaps this can be explained by the fact that, on the whole, students prefer to be approached rather than ask for help themselves. Although the investigation had too few students to make any definite conclusions, some tentative suggestions for future experiments might be made. Firstly, the motivation aspect should be studied thoroughly. This might explain why some of the potentially better students performed less well using CAI. Secondly, the design of CAI courses might be allowed to depend to some extent upon the students' attitude to such a course. Attitude questionnaires provide much useful information together with that from response files. Thirdly, an estimate of the real effect of an on-line processor should be studied more carefully. It is an accepted principle that hands-on experience is invaluable but justification of such a facility is important. Finally,

the psychological problems associated with the suggestion that the "help" facility be renamed the "advice" facility might be studied.

A rough estimate of the cost of the investigation suggested that it was rather higher than generally accepted. However, certain deficiencies in the operating system account for a greater part of this extra cost. A future study might consider what changes need to be made if a dedicated CAI system were to be used instead of a general-purpose time-sharing system. Naturally, certain features associated with large systems would be unavailable but the trade-off of these for decreased cost might possibly be tempting.

REFERENCES.

Adams, D.M. (1969). "An investigation into methods of presenting material for use in computer-assisted instruction." M.Sc. dissertation, Computing Laboratory, University of Newcastle upon Tyne, September, 1969.

Adams, E.N. (1967). "Reflections on the design of a CAI Operating System." IBM Research, Watson Research Centre, Yorktown Heights, N.Y., RC 1745, January 23, 1967.

Adams, E.N. (1969). "Technical considerations in the design of a CAI operating system." IBM Research, Watson Research Centre, Yorktown Heights, N.Y., RC 2557, July 29, 1969.

Avner, R.A. and Tanczar, P. (1969). "The TUTOR manual", Computer-based Education Research Laboratory, University of Illinois, January 1969.

Baker, J.D. (1965). "COBIS computer-based instruction system." Newsletter, Greater Boston Chapter of the Society for Programmed Instruction, 1, 4, 1965.

Bitzer, D.L. and Easley, J.A. (1965). "PLATO: A computer-controlled teaching system," in Sass and Wilkinson (Eds.), Computer Augmentation of Human Learning, Washington, Spartan Books, pp.89-103.

Clapp, D.J., Yens, D.P., Shettel, H.H. and Mayer, S.R. (1964). "Development and evaluation of a self-instructional course in the operational training capability query language for system 473L, USAF HQ". Air Force Electronic Systems Division, Decision Science Laboratory, Report No.ESD-TR-64-662, 1964.

Crowder, N.A. (1960) in "Teaching Machines and Programmed Learning", eds. Lumsdaine and Glaser, National Education Association, 1960.

Dean, P.M. (1969). "Preliminary Report on the Development of a Simplified System for CAI". Unpublished Report, IBM Education Research Department, San Jose, 1969.

Dean, P.M. (1970a). "Author's Guide to the COURSEMAKER System", IBM Education Research Department, San Jose, July, 1970.

Dean, P.M. (1970b). "Keypunch Operator's Guide to the COURSEMAKER System", IBM Education Research Department, San Jose, August, 1970.

Dean, P.M. (1970c). "The COURSEMAKER System", IBM Education Research Department, San Jose, August, 1970.

Dowsey, M.W. (1970a). "NUTS - User's Guide", Unpublished Report, Computing Laboratory, University of Newcastle upon Tyne, January 1, 1970.

Dowsey, M.W. (1970b). "Towards a true author entry system for CAI", Programmed Learning and Educational Technology, 7, 1 (January, 1970) pp 43-62.

Dowsey, M.W. (1970c). "A language to facilitate computer-aided instruction", Proceedings of I.E.E. Conference on Man-Computer Interaction, September, 1970, pp 72-76.

Easley, J.A. (1967). "Second midyear report for project SIRA", University of Illinois, Urbana, September, 1967.

Engvold, K.J. and Hughes, J.L. (1968a). "A general-purpose display processing and tutorial system", Technical Report TR 00.1694, IBM Systems Development Division, Poughkeepsie, N.Y., January 11, 1968.

Engvold, K.J. and Hughes, J.L. (1968b). "A multi-functional display system for processing and teaching", Proceedings of IFIP Congress 1968, North Holland Publishing Company, Amsterdam.

Feingold, S.L. (1967). "PLANIT - A Flexible Language Designed for Computer-Human Interaction", AFIPS Conference Proceedings, FJCC, 1967.

Fenichel, R.R., Weizenbaum, J. and Yochelson, J.C. (1970). "A Program to Teach Programming", CACM, 13, 3 (March 1970), pp 141-146.

Feurzeig, W. (1965). "The Socratic System: A Computer System to aid in Teaching Complex Concepts", Bolt, Beranek and Newman, Inc., Cambridge, Mass., 1965.

Feurzeig, W. and Papert, S.A. (1968). "Programming Languages as a Conceptual Framework for Teaching Mathematics", Proceedings of the NATO Conference on Computers and Learning, Nice, May, 1968.

Flanigan, L.K. (1968). "Introduction to PIL in MTS." Unpublished Report, Computing Centre, University of Michigan, Ann Arbor, May, 1968.

Frye, C.H. (1968). "CAI languages: capabilities and applications." Datamation, 14, 9 (September, 1968) pp 34-37.

Frye, C.H., Bennick, F.D. and Feingold, S.L. (1968). "Interim user's guide to PLANIT: the Author Language of the Instructor's Computer Utility." TM 3055/000/03, System Development Corp., Santa Monica, October 16, 1968.

Gilligan, J. (1969). "CAI Crawley". Technical Report, ICL Education Research Department, Loudwater, High Wycombe, September, 1969.

Gross, P.F., Cropley, A., Hebb, B., and Palmer, R. (1969). "APL and remote terminal usage for CAI". Paper presented at DATAFAIR 1969, Manchester, August 27, 1969.

Grubb, R.E. (1965). "The Effects of Paired Student Interaction in the Computer Tutoring of Statistics". Paper read at National Convention of the National Society for Programmed Instruction, Philadelphia, May, 1965.

Grubb, R.E. (1968). "Learner Controlled Statistics," Programmed Learning and Educational Technology, 4, pp 18 - 24, January, 1968.

Grubb, R.E. (1969). "A study of differential treatments in the learning of elementary statistics," paper presented at DAVI Conference, Portland, Oregon, April 28, 1969.

Grubb, R.E. and Selfridge, R.E. (1963). "Computer Tutoring in Statistics", Computers and Automation, 13, 3 (March, 1963).

Hansen, D.N. (1970). "Development processes in CAI problems, techniques and implications", Proceedings of a seminar on Computer-Based Learning Systems, NCET, March, 1970.

Hartley, J.R. and Sleeman, D.H. (1968). "Problem solving and simulation using a computer-based system", Proceedings of a NATO Conference Major Trends in Programmed Learning Research, Nice, May, 1968.

Hayward, P.R. (1968). "ELIZA scriptwriter's manual", Educational Research Centre, MIT, Cambridge, Mass., 1968.

Hesselbart, J.C. (1968) "FOIL: a file-oriented interpretive language", Proceedings of ACM National Conference, 1968.

Hickey, A.E. (1968). "Computer-assisted instruction: a survey of the literature. 3rd edition", ENTELEK Inc., Newburyport, Mass. October 1, 1968.

Homeyer, F.C. (1970). "Development and Evaluation of an Automated Assembly Language Teacher", Technical Report, CAI Laboratory, The University of Texas at Austin, 1970.

Hunt, E. and Zosel, M. (1968). "WRITEACOURSE: an educational programming language", AFIPS Conference Proceedings, FJCC, 1968.

IBM Corp. (1967). "The 1500 Instructional System: Introduction to Computer-assisted Instruction and System Summary", IBM Systems Development Division, Product Publications, San Jose, 1967.

IBM Corp. (1968). "The 1500 Instructional System: COURSEWRITER II, Author's Guide." Form Y26-1580-0, Special Systems Programming Documentation, San Jose, 1968.

ICL (1969) "The Brighton Project", Education Research Department, Loudwater, High Wycombe, 1969.

Keller, L. (1968) "Reference Manual: Course Author Language (CAL)", Computer Facility, University of California, Irvine, August, 1968.

Kerr, E.G., Ting, T.C. and Walden, W.E. (1969). "A control program for computer-assisted instruction on a general-purpose computer", Proceedings of ACM National Conference, 1969, pp 111-116.

Kopstein, F.F. (1969). "Computers and Instruction at Hum RRO", Educational Technology, 9, 7 (July, 1969) pp 25-28.

Kopstein, F.F. and Seidel, R.J. (1967). "Computer-administered instruction verses traditionally administered instruction: economics". Professional Paper 31-67, Hum RRO, Alexandria, Virginia, June, 1967.

Kristy, N.F. (1968). "Innovations of the Technomics 6700 System", Technomics, Inc., Santa Monica, May, 1968.

Lambert, P. (1968). "Computers and the Educational System", Aspects of Educational Technology, 2, 1968, Menthuen.

Lorton, P. and Slimick, J. (1969). "Computer-based instruction in computer programming - a symbol manipulation - list processing approach", AFIPS Conference Proceedings, FJCC, 1969, 35, pp 535-544.

Lyon, G. and Zinn, K.L. "Some procedural language elements useful in an instructional environment," Proceedings of a Seminar on Computer-based Learning Systems, NCET, March, 1970.

Maher, A. (1964). "Computer-Based Instruction (CBI): introduction to the IBM Research Project," IBM Research, Watson Research Centre, Yorktown Heights, N.Y., RC 1114, March 6, 1964.

Meadow, C.T., Waugh, D.W. and Miller, F.M. (1968). "CG-1, a course generating program for computer-assisted instruction", Proceedings of ACM National Conference, 1968, pp 99-110.

Mellan, I. (1936), in Journal of Experimental Education, 4, March, 1936.

National Council for Educational Technology (1968). "Computer Based Learning Systems", Report of a working party of the National Council for Educational Technology, November, 1968.

Pask, G. (1959). "The Teaching Machine", The Overseas Engineer, February, 1959, pp 231-232.

Perstein, M.H. (1966). "Grammar and lexicon for basic JOVIAL", System Development Corp., Santa Monica, Technical Memorandum, TM-555/005/00, May 10, 1966.

Philco-Ford Corp. (1970a). "INFORM Author Reference Manual", Communications and Technical Services Division, Willow Grove, Pennsylvania, February, 1970.

Philco-Ford Corp.(1970b). "Project GROW User's Manual", Communications and Technical Services Division, Willow Grove, Pennsylvania, April, 1970.

RCA (1967). "Instructional 70, General Information Manual", RCA Instructional Systems Division, Palo Alto, California, 1967.

RCA (1968). "Instructional 71, General Information Manual", RCA Instructional Systems Division, Palo Alto, California, 1968.

Rath, G.J., Anderson, N.S. and Brainerd, R.C. (1960). "The IBM Research Centre Teaching Machine Project", in Galanter (Ed.), Automatic Teaching, The State of the Art, New York, Wiley, 1960.

Ruans, D.G. (1963). "An Information Systems approach to Education", System Development Corp., Technical Memorandum, TN-1495, 1963.

Schramm, W. (1964). "The Research on Programmed Instruction, an Annotated Bibliography", U.S. Office of Education Bulletin No. 35, 1964.

Schurdak, J. (1967). "An Approach to the use of computers in the instructional process and an evaluation", IBM Research, Watson Research Centre, Yorktown Heights, N.Y., RC 1432, 1967.

Silvern G.M. and Silvern L.C. (1966a). "Computer-assisted instruction: Specification of attributes for CAI programs and programmers", Proceedings of ACM National Conference, 1966, pp 57-62.

Silvern, G.M. and Silvern, L.C. (1966b). "Programmed Instruction and computer-assisted instruction....an overview", Proceedings of the IEEE, 54, December, 1966, pp 1648-1655.

Skinner, B.F.(1954). "The Science of Learning and the Art of Teaching", Harvard Educational Review, 24, Spring, 1954, pp 86-97.

Sleeman, D.H. and Hartley, J.R. (1968). "The design and some possible uses of a computer-assisted learning system", Aspects of Educational Technology, 2, 1968, pp 537-542.

Starkweather, J.A. (1968). "PILOT User's guide: a conversational computer language", University of California Medical Centre, San Francisco, December 1, 1968.

Starkweather, J.A. and Turner, W. (1966). "COMPUTEST II-D: a programming language for computer-assisted instruction, testing and interviewing", Computer Centre, University of California, San Fransisco, November, 1966.

Stolurow, L.M. (1965a). "Model the Master Teacher or Master the Teaching Model", in Krumboltz (Ed.), Learning and the Education Process, Chicago, Rand McNally and Co., 1965, Chapter 9, pp 223-247.

Stolurow, L.M. (1965b). "Computer-based instruction", University of Illinois, Training Research Laboratory, Contract NONR 3985(04), Report No.9, 1965.

Summers, R.C., Wood, J.R., Citron, J.P. and Bray, R.R. (1967). "Design of a Supervisor for Interactive Applications", IBM Los Angeles Scientific Centre, 1967.

Swets, J.A. and Feurzeig, W. (1965). "Computer-Aided Instruction", Science, 150, 3696 (October 29, 1965) pp 572-576.

Tonge, F.M. (1968). "Design of a programming language and system for computer-assisted learning", Proceedings of IFIP Congress 1968, North Holland Publishing Company, Amsterdam.

Uhr, L. (1969). "Teaching machine programs that generate problems as a function of interaction with students", Proceedings of ACM National Conference, 1969, pp 125-134.

University of Michigan (1967). "Michigan Terminal System", User's manual, Computing Centre, University of Michigan, Ann Arbor, 2nd. edition, December 1, 1967.

Uttal, W.R. (1962). "My Teacher has Three Arms!!!", IBM Research, Watson Research Centre, Yorktown Heights, N.Y., RC 788, September, 1962.

Weizenbaum, J. (1966). "ELIZA - a computer program for study of natural language communication between man and machine", CACM, 9, 1 (January, 1966) pp 36-45.

Weizenbaum, J. (1967). "Contextual understanding by computers", CACM, 10, 8 (August, 1967), pp 474-480.

Winkler, C.E. (1968). "Computer-assisted instruction as an information retrieval public utility", Proceedings of the American Society for Information Science, 5, (1968), pp 169-173.

Wodtke, K.W., Mitzel, H.E., and Brown, B.R. (1965). "Some preliminary results on the reactions of students to computer-assisted instruction", Pennsylvania State University, Paper for a symposium, Systematic Instruction, APA Convention, 1965.

Zinn, K.L. (1965). "Functional specifications for computer-assisted instructional systems", in Goodman (Ed.) Automated Education Handbook, Detroit, Automated Education Centre, 1965, IV, A21-32.

Zinn, K.L. (1967). "Computer Assistance for Instruction: A Review of Systems and Projects", in Bushnell and Allen (Eds.) "The Computer in American Education", John Wiley, 1967.

Zinn, K.L. (1968). "Programming conversational use of computers for instruction", Proceedings of ACM National Conference, pp 85-92.

Zinn, K.L. (1970). "A comparative study of languages for programming interactive use of computers in instruction", Proceedings of a seminar on Computer-Based Learning Systems, NCET, March, 1970.

APPENDIX A.

Newcastle University Teaching System (NUTS).

User's Guide

September 30 1970.

CONTENTS.

Introduction

Newcastle University Teaching System is a system designed to permit natural communication with a computer by providing the facilities for a conversational dialogue to take place between a person and a computer. One class of user called an author is able to build lessons which will form part of a course which, in turn, is used for the purpose of teaching another class of user called a student.

The basis of the system is the command language, all of which may be used by an author but only a subset of which may be used by a student.

The author builds his lessons by using the author language. This is a high-level programming language which enables him to present subject material to the student, ask a question then specify combinations of keywords and/or values to be sought in specified degrees of accuracy in the student's ensuing reply. Depending upon the nature of this reply, and, perhaps, any previous response in the student's history of the lesson, the author then directs the student to follow a particular path through the rest of the course.

The student may come to the terminal for a session with one or more of the courses whenever it is convenient, for the author usually writes each lesson in the course in segments and at the start of each session the student re-starts at the beginning of the segment he was in at the end of the previous session. He can terminate his session whenever the keyboard is unlocked to him. Alternatively, the author may allow the student complete freedom to move about the course at will by specifying points in a lesson to which the student may proceed by using predetermined responses.

For his own use, the author retains information about each student using his courses in what are called response files.

As an aid to answering any question, a simple calculating language (which generates programs) and a sequence controlled desk machine are available to the student at any time, i.e. not only during a reply to a question but in the command language.

Also included is PIL, a simple language, easy to learn, but with such powerful features as string manipulation, extended I/O, etc. PIL possesses a powerful "direct" mode which has superceded the sequence controlled desk machine. "Indirect" statements may be stored from one call of the interpreter to the next.

Apart from during a session when a course is being taken, all input to the system is of entirely free format, embedded blanks being allowed.

CONVENTIONS

The notational conventions described here are used in the command and language source statement format illustrations to explain how each operand is to be written. To facilitate the representation of the statements in the format illustrations, three metasymbols are used as follows:-

braces { } To enclose and therefore delimit syntactical units (one or more operands) that may be repeated. Alternatives may be indicated by aligning the choices vertically within the braces: {A / B / C}

brackets [ ] (a) to enclose and thus delimit alternatives,

(b) to enclose and thus delimit optional names and/or

operands within the appropriate fields. Stacked items within the optional syntactical unit show alternatives.

ellipses ... To indicate that the preceding syntactical unit may be repeated one or more times. Should a system limit to the number of repetitions permitted exist, this will be given in the operand list that follows the format illustration.

THE COMMAND LANGUAGE

INTRODUCTION

The command language is designed for users who communicate with NUTS while it is executing operations for them. This mode of operation is called conversational, since the user remains on-line to NUTS, engaging in a dialogue with it.

The command language also serves as the job control language for operations that do not require a dialogue with the user; i.e. operations submitted to the system for execution without user monitoring. This second mode of operation, in which the user is not on-line to the system, is described as the non-conversational mode. Both modes use the same command language, except that some commands are only available in conversational mode.

TASK INITIATION

The user initiates his task by firstly signing on to Michigan Terminal System. When his signon has been accepted and he is prompted by a number sign (#), he enters the MTS command "ØSOURCE NUTS". This activates NUTS and the user is in command mode after the appearance of "*".

REQUESTS FOR NEXT COMMAND

The system informs the user that it is ready to accept his next command by printing an asterisk, "*", then unlocking the keyboard after giving a carriage return line feed.

ENTERING COMMANDS

Every command entered from a terminal keyboard starts on the new line after the prompt, "*". The user may employ a completely free format, with as many embedded blanks as he requires in any position. The only restriction is that commands may not exceed one line and this line must not exceed 80 characters. Truncation of extra characters occurs. The end of the command line is indicated by pressing the RETURN key.

If commands are entered from cards, each must start on a new card, but otherwise the above rules apply.

COMMAND FORMAT

The general format of the command language statements is:

OPERATION	SEPARATOR	OPERAND
command name	a comma; blank if and only if operand field is blank.	one or more operands delimited by commas; field may be blank.

The operand field is separated from the operation field by a comma. The operand field itself may be blank for certain commands or may contain several operands separated from one another by commas.

COMMAND DESCRIPTIONS

The command descriptions are in alphabetical order; each has the following arrangement:

1. the command name.
2. a brief statement of the command's functions.
3. when and by whom it may be used.
4. the command format is illustrated and each operand is described.
5. a description of the command is then given, discussing what the command does from a user's standpoint, and telling about its restrictions and limitations.
6. finally, one or more examples are given to show exactly how the command is used.

Here is a list of special terms used in command descriptions.

arrname	is a valid array name consisting of an ampersand (&) followed by one through six letters.
csname	is a valid course name consisting of between one and five letters.
digit	is a valid digit, 0 through 9.
element number	is an integer 1 through 100.
finline	is the finishing line of a sequence of line numbers and is an integer greater than zero but less than or equal to 999/99999 for programs/lessons.
incr	is the increment between successive line numbers. Its bounds are as for finline.
line number	is an integer which is valid if between 1 and 999/99999 for programs/lessons.
lname	is a valid lesson name consisting of the constituent course name's one through five letters followed by a digit.
name	is a string of characters inadvertently entered.
prname	is a valid program name consisting of between one and six letters.
rfile	is a valid response file name consisting of the course name's one through five letters followed by a number sign (#) then a digit.
segment number	is an integer 1 through 99.
stline	is the starting line of a sequence of line numbers. Its bounds are as for finline.
userid	is a valid user identification sequence consisting of two letters followed by either a letter and a digit or a digit and a letter.
varname	is a valid simple variable name consisting of one through six letters.

COMMANDS

The commands are presented in alphabetical order for ease of reference.

BUILD COMMAND

1. BUILD.

2. This command is used to translate a series of author language source statements into intermediate code. If errors are detected, appropriate diagnostic error messages are produced but if successful a lesson is "built" and may be subsequently called upon. The source statements may be entered in full or in part within the BUILD command or, if desired, may already exist in full in the lesson file.

3. On terminal and in batch. Authors only.

4. BUILD, lname [{,N[,stline,incr]} / {,M}]

4.1 lname — If the second operand is "M" or blank then the lesson file must already exist; if "N", the lesson should not yet exist.

4.2.1 N — is entered if a new lesson is to be created. Prompting then occurs for each source statement, which is processed immediately, before a prompt is given for the next line.

4.2.2 M — is entered if modifications are to be made to the lesson source statements before translation is attempted.

4.2.3 (blank) — specifies that translation only will take place. No chance is given to the author to enter source statements prior to processing.

4.3 stline — is entered only if "N" is the second operand. It specifies the line number to be given to the first statement prompted for. Limits are 1 and 99999. If absent, the default value is 10.

4.4 incr — is also entered only if "N" is the second operand. It specifies the increment between any successive line number prompts. Limits are also 1 and 99999. If absent - it may be absent if and only if "stline" is absent - the default value is also 10.

5. It is obvious that there are three modes of operation of the command BUILD.

5.1 mode 1

5.1.1 The author may wish to create a new lesson and enter his source statements within the command. To do this he specifies N and optionally gives the starting line number for the statements in the lesson and the increment with which the line number will increase on successive prompts. The system prompts "line number>". The author then enters his source statement which is immediately parsed and intermediate code generated.

5.1.2 If this is successful, the line number is incremented and the system prompts once more.

5.1.3.1 If unsuccessful and from a terminal, an appropriate diagnostic message is returned, together with a prompt to make a modification. This takes the form ">". At this point the terminal author may modify any previous line, delete any previous line or insert any line to his lesson, all in the same format.

5.1.3.2 This format is:

line number, line contents

where the line number must be a valid integer between 1 and 99999 and the line contents contain the modified line, blank as necessary for deletion. The comma is mandatory. As the maximum length of line allowed by the author language processor is 80, in the case of a modification, the length of an actual line will be slightly less. One consideration when making modifications in this way is as follows: if the author corrects a line immediately he is told it is incorrect, then the processor may continue with one and only one pass, no re-translation of the whole lesson would be necessary; otherwise, in the event of a modification of a different line from that given in the current incremented line number prompt, the line is checked syntactically within itself, but without reference to the rest of the lesson. The whole lesson is re-translated at a later stage. When the author has completed his modifications, he simply presses RETURN to go back to the incremented line number prompt.

5.1.3.3 If unsuccessful and in batch, the incremented line number prompt continues as if it had been correct.

5.1.4 On entry of the statement END, the processor completes the label chaining, etc.

5.1.5 Then, if from a terminal, the author is given the option for a complete or part source listing, whether he wishes to make further modifications, and whether he wishes to continue processing. In batch none of these options is available. Finally, if the translation has been successful, the command terminates, but if unsuccessful then the author from a terminal has a further option for modifications but in batch, termination occurs with an appropriate message.

5.2 mode 2 The author may wish to re-translate a previously existing lesson but first enter some modifications. To do this he specifies M. The modifications are entered in the format specified in 5.1.3.2. If an incorrect source statement is entered for a modification, then, from a terminal it may itself be modified but, in batch, modifications cease at the point. Of course, this will produce invalid commands in the command sequence, all of which will subsequently be ignored. When modifications have all been entered, execution continues as in 5.1.5.

5.3 mode 3 The author may wish to re-translate only a previously existing lesson. He leaves the second operand of the command blank. If the translation is unsuccessful, then the terminal author has the chance to enter modifications. In batch, termination occurs with an appropriate message.

6. Examples.

6.1 mode 1

```
user :  BUILD,HISTØ,N,1ØØ,2Ø
sys  :  FILE "HISTØ    " HAS BEEN CREATED
sys  :  ENTER STATEMENTS
sys  :  100 >
user :  (enters his source line)
sys  :  120 >
users:  (enters next source line)
  :
  :
ETC.
```

A new lesson named "HISTØ" has been created and lines entered by prompting at 100, increasing by 20 each time.

```
user :  BUILD,GEOGØ,N
sys  :  FILE "GEOGØ" HAS BEEN CREATED
sys  :  ENTER STATEMENTS
sys  :  10 >
user :  (enters his source line)
sys  :  20 >
user :  (enters next source line)
  :
  :
 ETC.
```

A new lesson named "GEOGØ" has been created but this time the starting line prompt and the increment have been allowed to default to 10.

6.2 mode 2

```
user :  BUILD,ARITH2,M
sys  :  ENTER MODIFICATIONS
sys  :  >
user :  (enters source line with line number)
sys  :  >
user :  (enters source line with line number)
  :
  :
 ETC.
```

A previously existing lesson "ARITH2" is to be re-translated but modifications are to be entered first.

6.3 mode 3

```
user :  BUILD,ALGEB9
sys  :  END OF 'BUILD'
sys  :  *
```

A previously existing lesson "ALGEB9" is to be re-translated only.

(The example assumes an error-free translation)

CALC COMMAND

1. CALC.

2. This command is used to translate a series of calculating language source statements into intermediate code. If errors are detected, appropriate diagnostic messages are returned, but if successful the user runs the program which is also stored for future use. The source statements may be entered in full or in part within the CALC command, or, if desired, may already exist in full in the program file.

3. On terminal and in batch. Authors and students.

4. CALC, prname $\left[\left\{\begin{matrix},N[,stline,incr]\\,M\end{matrix}\right\}\right]$

4.1 prname — must be a valid program name. If the second operand is "N" or blank then the program file must already exist; if "N" the program should not yet exist.

4.2.1 N — is entered if a new program 'prname' is to be created. Prompting then occurs for each source statement, which is processed immediately, before a prompt is given for the next line.

4.2.2 M — is entered if modifications are to be made to the program source statements before translation is attempted.

4.2.3 (blank) — specifies that translation only will take place. No chance is given the user to enter source statements prior to processing.

4.3 stline — is entered only if "N" is the second operand. It specifies the line number to be given to the first statement prompted for. Limits for "stline" are 1 and 999. If absent when "N" has been entered, the default value is 10.

4.4 incr. — is also entered only if "N" is the second operand. It specifies the increment between any successive line number prompts. Limits are 1 and 999. If absent - it may be absent if and only if "stline" is absent - the default value is also 10.

5. As in the BUILD command, there are three modes of operation of the command CALC.

5.1 mode 1

5.1.1 The user may wish to create a new program and enter his source statements within the command. To do this, he specifies N and optionally gives the starting line number for the statements in the program and the increments with which the line number will increase on successive prompts. The system prompts "line number>". The user then enters his source line which is immediately parsed, and intermediate code generated.

5.1.2 If this is successful, the line number is incremented and the system prompts once more.

5.1.3.1 If unsuccessful and from a terminal, an appropriate diagnostic message is given, together with a prompt to make a modification. This takes the form ">". At this point the terminal user may modify any previous line, delete any previous line or insert any line to his program, all in the same format.

5.1.3.2 This format is:

line number, line contents

where the line number must be a valid integer between 1 and 999, and the line contents contain the modified line, blank, if necessary, for deletion. The comma is mandatory. As the maximum length of line allowed by the calculating language processor is 80, in the case of a modification, the length of an actual source line will be slightly less than 80. One consideration when modifying lines in this way is as follows: if the user corrects a line immediately he is told that it is incorrect, then the processor may continue with one and only one pass; no retranslation of the whole program would be necessary; otherwise, in the event of the modification of a different line from that given in the current incremented line number prompt, the line is checked syntactically within itself, but without reference to the rest of the program. When the author has completed his modifications he simply presses RETURN to go back to the incremented line number prompt.

5.1.3.3 If unsuccessful and in batch, the incremented line number prompt continues as if the line had been correct.

5.1.4 On entry of the statement END, the processor completes the label chaining, etc.

5.1.5 Then, if from a terminal, the user is given the opportunity to acquire a complete or part source listing, to make further modifications and to continue processing. In batch, none of these options is available. Finally, if the translation has been successful, the program runs with the user supplying data as requested. If unsuccessful, the terminal user has a further chance for supplying modifications, but in batch, termination occurs with an appropriate message.

5.2 mode 2 The user may wish to re-translate a previously existing program but first enter some modifications. To do this he specifies M. The modifications are entered in the format specified in 5.1.3.2. If an incorrect source statement is entered for a modification then from a terminal, it may itself be modified but in batch, modifications cease at that point. Of course this will produce invalid commands in the command sequence, all of which will subsequently be ignored. When modifications have all been entered, execution continues as in 5.1.5.

5.3 mode 3 The user may wish to re-translate only a previously existing program. He leaves the second operand of the command blank. If the translation is unsuccessful, then the terminal user has the opportunity to enter modifications. In batch, termination occurs with an appropriate message.

6. Examples.

6.1 mode 1

```
user :  CALC,CUBICS,N,50,15
sys  :  FILE "CUBICS  " HAS BEEN CREATED
sys  :  ENTER STATEMENTS
sys  :  50 >
user : (enters his source line)
sys  :  65 >
user : (enters next source line)
  :
  :
 ETC.
```

A new program named "CUBICS" has been created and the lines to be entered prompted for starting at 50 and increasing by 15.

```
user :  CALC,SUM,N
sys  :  FILE "SUM  " HAS BEEN CREATED
sys  :  ENTER STATEMENTS
sys  :  10 >
user :  (enters his source line)
sys  :  20 >
user :  (enters next source line)
  :
  :
 ETC.
```

A new program named "SUM" has been created but this time the starting line prompt and the increment have been allowed to default to 10.

6.2 mode 2

```
user :  CALC,JACOBI,M
sys  :  ENTER MODIFICATIONS
sys  :  >
user :  (enters source line with line number)
sys  :  >
user :  (enters line with line number)
  :
  :
 ETC.
```

A previously existing program "JACOBI" is to be re-translated but modifications are to be entered first.

6.3 mode 3

```
user :  CALC,STD
sys  :  (prompts for required inputs)
  :
  :
 ETC.
```

A previously existing program "STD" is to be re-translated only. (The example assumes an error-free translation and so the program runs).

CAT COMMAND

1. CAT.

2. This command tells the user which files he possesses at that point in time.

3. On terminal and in batch. Authors and students.

4. CAT [{,L / ,P / ,R}]

4.1.1 (blank) - student is told which programs he owns (this is the only valid way a student may use the command)

author is given a list of lessons, programs and response files he owns.

4.1.2 L (author only); a list of lessons is given.

4.1.3 P (author only); a list of programs is given.

4.1.4 R (author only); a list of response files is given.

5. Each list is given an appropriate heading. If it is null, therefore, only the heading will appear. In the case of lessons, the word "RELEASED" will appear against the lesson name if it has been released for general use. Only the author is permitted to see which response files he owns.

6. Examples.

6.1 author

```
user : CAT
sys  :      PROGRAMS
sys  : CUBICS
sys  : SUM
sys  :
sys  :      LESSONS
sys  : MATHSØ
sys  :
sys  :      RESPONSE FILES
sys  :
sys  : END OF 'CAT'
sys  : *
```

All the author's files are listed. He has two programs, one lesson but no response file.

```
user : CAT,L
sys  :     LESSONS
sys  : HISTØ  RELEASED
sys  : HIST1
sys  : GEOGØ
sys  :
sys  : END OF 'CAT'
sys  : *
```

The author has requested only the list of his current lessons. He has three, one of which has been released for general use.

6.2 student

```
user : CAT
sys  :      PROGRAMS
sys  : JACOBI
sys  : STD
sys  : SQUARE
sys  :
```

```
sys  :   END OF 'CAT'
sys  :  *
```

The student is given a list of the programs he possesses. Even though he may own some response files implicitly, that they exist is of no useful value to him so he is not told of their existence.

COPY COMMAND

1. COPY.

2. This command allows the user to copy an existing file into another file, which may already exist or need to be created.

3. On terminal and in batch. Authors and students.

4. COPY, file1, file2.

   4.1 file1 - must be a valid lesson (authors only) or program which exists.

   4.2 file2 - must be a valid name of a file of the same type as "file1". It may or may not exist already.

5. The source statements only are copied from "file1". To run either the lesson or program subsequently, it must be translated.

6. Examples.

```
user : COPY,ALGEB1,MATHS7
sys  : *
```

The source statements of lesson "ALGEB1" are copied into lesson "MATHS7", which already exists.

```
user : COPY,CUBICS,SOLVE
sys  : FILE "SOLVE" HAS BEEN CREATED
sys  : *
```

The source statements of program "CUBICS" are copied into program "SOLVE", which has first been created.

COURSE COMMAND

1. COURSE.

2. This command allows the user to take part in a course, if it exists and has been translated successfully. In the case of students, the constituent lesson 0 (at least) must have already been released for general use.

3. On terminal only. Authors and students.

4. COURSE, csname

   4.1 csname is a valid course which exists. Authors may use the command to check out the constituent lessons of the course before satisfying themselves that these are fit for release. Students may use the command only if lesson 0 of the course has already been released.

5. The system first decides whether the user is an author or a student (one author may be acting as a student to another author); also if the course exists.

5.1 author: the author is asked which response file he wishes to use for this particular run of the course. He has ten response files available to him per course - they are known by digits 0 to 9 to him. They are available so that the author may explore the many possible routes through his course and correct inconsistencies, if necessary, before allowing students to use it.

The system then tells him at what segment he will begin and in what lesson. If he wishes to carry on from this specified point in the course, the author just presses RETURN; otherwise he is allowed to specify the segment and the lesson from where he will re-commence.

If the lesson is available for use, i.e. if it has been translated successfully, and the segment exists, then the course is taken up at that point. There will only be any doubt about the existence of a segment if the author has specified his own re-entry point and not just carried on from where he left off previously.

5.2 student: if the course specified by the student is available to him, then he usually re-commences at the beginning of the segment he was in when he terminated his last course session. However, the author may overrule this convention if he wishes the student to have complete freedom within a course.

5.3 Once inside the instructional material, the user will be presented with various facts and from time to time asked questions on the subject content of the course. Thus, the keyboard is unlocked to the user when he is expected to reply to a question. His response may be up to five lines long, providing the continuation character, "-", is used at the end of the previous line all the way. The length of each line must not exceed 80 characters, including the continuation character.

The only other time that the keyboard is unlocked to the user is when the course author has specified a pause at that point. The system prompts with a ":", followed by a carriage return line feed, and then the user may restart when he is ready by simply pressing RETURN only.

Whenever the keyboard is unlocked, the user may perform certain tasks other than simply making a reply in the case of a question, or proceeding in the case of a pause.

5.3.1 ?END — if the user enters "?END" in either instance, the course is terminated at that point and he is returned to the command language prompt.

5.3.2 ?F — during any segment, an author may specify certain parts of the lesson to which the student will be sent if he enters a predetermined non-solution response when the keyboard is unlocked to him. This the author does by use of a CTRL statement. If a user enters "?F" then he will be sent to the point described by the first parameter of the CTRL statement. If the author has not specified a CTRL statement within the segment, then the "?F" is ignored, i.e. in the case of a question, "?F" is assumed to indicate the student's answer; in the case of a pause, "?F" will cause a restart. The reason the letter F was chosen was that the first parameter might be used for a forward skip; however, this is completely arbitrary.

5.3.3 ?B — this is exactly the same as for "?F" except that the student will be sent to the point described by the second parameter of the CTRL statement, if it has been used. The letter B was intended to suggest a backward skip.

5.3.4 ?S — as for "?F" but the point is given by the third parameter of CTRL, if it has been used. The letter S was intended to suggest a skip to the subject outline.

5.3.5 ?G — as for "?F" but the point is given by the fourth parameter of CTRL, if it has been used. The letter G was intended to suggest a skip to a glossary.

5.3.6 ?CALC — this may only be entered in reply to a question. It indicates that the student wishes to write or modify a program in the calculating language. Of course, the usual reason for his wanting to do this is that it will help him answer the question. The system asks him either to enter the parameters (see command CALC) or simply press RETURN if he wishes to cancel the request. When he is finished using his program or, indeed, when he has cancelled the request, the student is reminded to answer the question.

5.3.7 ?PROG — this may only be entered in reply to a question. It indicates that the student wishes to run a program in the calculating language. The system asks him either to enter the filename of the program (see command PROG) or simply press RETURN if he wishes to cancel the request. On completion, the student is reminded to answer the question by the system.

5.3.8 ?DESK — this may only be entered in reply to a question. It indicates that the student wishes to use the desk calculator. On exit, he is reminded to answer the question.

5.3.9 ?PIL — this may only be entered in reply to a question. It indicates that the student wishes to use the PIL interpreter. On exit, he is prompted to answer the question.

6. Examples.

6.1 Author

```
user :  COURSE,MATHS
sys  :  ENTER DIGIT TO DENOTE WHICH RESPONSE FILE TO USE
user :  2
sys  :  YOUR STARTING POSITION WILL BE LESSON 4 SEGMENT 13.
sys  :  IF YOU WISH TO CONTINUE FROM THERE JUST PRESS RETURN.
sys  :  OTHERWISE ENTER 'LESSON NO., SEG. NO.' TO RESTART.
user :  3,27
sys  :  (course continues)
```

The author continues to check out course 'MATHS' using response file MATHS#2 but restarting from lesson 3, segment 27 instead of lesson 4 segment 13.

6.2 Student

```
user :  COURSE,HIST
sys  :  (course continues)
```

The student merely declared his intention to use the course; where he recommences is determined already by the author.

DESK COMMAND

1. DESK.

2. This command allows the user access to the sequence controlled desk calculator.

3. On terminal only. Authors and students.

4. DESK.

INSERT COMMAND

1. INSERT.

2. This command allows users to create new lessons or programs or update existing ones by entering source statements into specified lines of the files. No syntax checking is carried out.

3. On terminal and in batch. Authors and students.

4. INSERT, filename, $\left\{\left\{\begin{matrix}L\\P\\U\end{matrix}\right\}[,stline,incr]\right\}$

4.1 filename — must be either a valid lesson name or a valid program name. "L" requires that it is a lesson which does or does not already exist, "P" a program which does or does not already exist. "U" allows either so long as the file does exist.

4.2.1 L — is entered if a new lesson is to be created or if an existing lesson is to be updated by successive line number prompts.

4.2.2 P — is entered if a new program is to be created or if an existing program is to be updated by successive line number prompts.

4.2.3 U — is entered if a previously existing file is to be updated but in such a way that the line number as well as the line is entered after a prompt.

4.3 stline — is entered only if either L or P is the second operand. It specifies the line number to be given to the first statement prompted for. Limits are 1 and 99999 for L but 1 and 999 for P. If absent when either L or P is given, the default value is 10.

4.4 incr — is also entered only if either L or P is the second operand. It specifies the increment between any successive line number prompts. The limits are as for "stline". If absent - it may be absent if and only if "stline" is absent - the default value is also 10.

5. It is obvious that there are two modes of operation of the command INSERT.

5.1 mode 1 The user may wish to enter his modifications in a rigid pattern of line numbers, for example, thirty lines each ten lines apart, (whether the file into which he is to insert exists or not is of little consequence except that he is warned when it does already exist). To do this he specifies L or P depending on the type of file and optionally gives the line number for the first statement he is to enter and the increment with which the line number will increase on successive prompts. The system prompts "line number>". The user then enters his source statement and presses RETURN. Prompting of line number then occurs until the user wishes to enter no more lines. At that point he simply presses RETURN in reply to the prompt and the command terminates.

The user may override the line number prompt at any time he wishes to modify or delete any other existing line or insert one out of sequence by entering his line in the format:

% line number, line contents

where the line number is a valid one depending on whether L or P was specified, and line contents are the actual insertion required.

After this particular line has been entered, the system prompts the same line number again that was overridden.

5.2 mode 2 If the modifications to be made seem randomly scattered about the file, then the user will not want a rigid line number prompt. Instead he will supply the line number to each modification. To do this he specifies U as the second operand, an action which requires that the file must already exist. The system prompts simply with ">" to which the user replies by entering his modification in the form

line number, line contents

where, as before, the line number is a valid one depending on whether the file being updated is a lesson or a program, and line contents are the actual insertion desired.

On each successive return from the user, the system gives a further prompt, to which the user replies RETURN only if he wishes to discontinue entering modifications.

It is important to note that updating a previously translated file via the INSERT command destroys the intermediate code and necessitates a re-translation before further use.

6. Examples.

6.1 mode 1

```
user :  INSERT,MATHSØ,L,25,50
sys  :  FILE "MATHSØ" HAS BEEN CREATED
sys  :  25 >
user :  (enters source line)
sys  :  75 >
user :  (enters source line)
  :
  :
 ETC.
```

An author has created a new lesson "MATHSØ" and wishes to insert source lines commencing at line 25 and thereafter increasing by 50 on successive prompts.

```
user : INSERT,ROOT,P
sys  : WARNING. PROGRAM 'ROOT  ' ALREADY EXISTS
sys  : 10 >
user : (enters source line)
sys  : 20 >
user : (enters source line)
  :
  :
 ETC.
```

An author or student wishes to insert source lines in a lesson "ROOT" which already exists. The starting line number and increment have been allowed to default to 10.

6.2 mode 2

```
user :  INSERT,LATIN5,U
sys  :  >
user :  (enters source line together with line number)
sys  :  >
  :
  :
 ETC.
```

An author wishes to update an existing lesson "LATIN5" but each modification he enters will contain a line number.

LESSON COMMAND

1. LESSON.

2. This command is available to inform authors which lessons are currently released for general use and who are their authors.

3. On terminal and in batch. Authors only.

4. LESSON.

5.

6. Example.

```
user :  LESSON
sys  :   COURSE     LESSON NUMBER     AUTHOR
sys  :   MATHS             0           CLC7
sys  :   PIL               0           DLCO
sys  :   MATHS             1           CLC7
sys  :   MATHS             5           CLC7
sys  :   PDE               0           QCC9
sys  :  END OF 'LESSON'
sys  :  *
```

Five lessons from three different courses here have been released so far.

LIST COMMAND

1. LIST.

2. This command gives the user a current listing of the file he specifies.

3. On terminal and in batch. Authors and students.

4. LIST,filename[,{(stline,finline) / (stline)}]

   4.1 filename — is the name of either a lesson or a program which exists.

   4.2.1 (blank) — specifies that a listing of all the lesson or program is required.

   4.2.2 stline — is the line number of the first line of the listing. If this does not exist, then the first line will be the one with the next greater line number than stline.

   4.3 finline — is the line number of the last line of the listing. If this does not exist, then the last line will be the one with the next smaller line number than finline.

5. The user is supplied with a current listing which contains all the lines in the specified part of the file. If the specified part is empty, then the command simply gives back a heading followed by an end of command message.

6. Example.

   The following NUTS dialogue shows how the command LIST may be used in three forms, using the same program, SQROOT, each time.

```
user :  LIST,SQROOT
sys  :          CURRENT LISTING
sys  :     50   1)READ(X)
sys  :    100   IF(X<LT>0)JUMP>2
sys  :    150   TYPE('X='1<X>3,10,5'SQUARE ROOT OF X='15<#SQT(X)>33,10,5)
sys  :    200   JUMP>1
sys  :    250   2)STOP
sys  :    300   END
sys  :  END OF 'LIST'
sys  :     *
user :  LIST,SQROOT(170,250)
sys  :          CURRENT LISTING
sys  :    200   JUMP>1
sys  :    250   2)STOP
sys  :  END OF 'LIST'
sys  :     *
user :  LIST,SQROOT(200)
sys  :          CURRENT LISTING
sys  :    200   JUMP>1
sys  :    250   2)STOP
sys  :    300   END
sys  :  END OF 'LIST'
sys  :     *
```

PIL COMMAND

1. PIL

2. This command places the PIL interpreter at the user's disposal.

3. On terminal and in batch. Authors and students.

4. PIL

5. The user has available all "indirect" statements from previous usage of the interpreter. Upon completion, entry of "MTS" returns the user to command mode.

PROG COMMAND

1. PROG.

2. This command is used to run any program in the calculating language which has been translated successfully. All inputs are prompted for by variable name and run time diagnostic messages are given where appropriate.

3. On terminal and in batch. Authors and students.

4. PROG, prname

4.1 prname must be the valid name of a program which exists and has been successfully translated previously by the CALC command.

5. The only messages to be given back apart from output results and output text are run time diagnostics and prompts resulting from the source statement READ.

5.1 The letter C before the diagnostic message indicates to the user that it is a controller failure. The run terminates immediately.

5.2 There are three types of READ statement, all of which generate different prompts to the user.

5.2.1 When a simple variable is to be read in, the system prompts with the name of the variable. The user is required to enter one valid constant only, followed by RETURN. Entry of a real constant when an integer constant was asked for causes the printing of a warning message and rounding occurs.

5.2.2 When an array element is to be read in, the system prompts with the actual element number required; that is, it evaluates any expression representing the array element, then prints out the array name followed by the element number value, contained in parentheses. As above, the user is required to enter one constant only, followed by RETURN. Also, this may be rounded, as appropriate.

5.2.3 When a sequence of array elements is to be read in, the system prompts with the array name and the actual element number from whence the constants are required; that is, it evaluates the simple variable which defines the starting element, if used instead of a constant. The user enters as many constants as he desires and these are stored in consecutive elements of the array. He separates them by a comma but terminates the sequence with a semi-colon. As the user may only enter 80 characters in any one line, he may continue on another line by neglecting to end the current line with a semi-colon, e.g. by a space or a comma. The system will then prompt for the next element in sequence. The user then either carries on or simply terminates with a semi-colon. If an error is detected in a line of constants, the user is asked to re-input the current line. Entry of commas only in sequence will cause zeros to be entered in the corresponding array elements. Also, rounding may occur as appropriate, but with warning.

6. Example.

```
user :  PROG,CUBICS
sys  :  ENTER DATA FOR VARIABLE LENGTH
user :  25.7
sys  :  W REAL CONSTANT INSTEAD OF INTEGER - ROUNDING OCCURS
sys  :  ENTER DATA FOR ARRAY ELEMENT  &TIME (16)
user :  1.384
sys  :  ENTER DATA FOR ARRAY &NUM STARTING AT ELEMENT 10
user :  5,8,3.2,10
sys  :  W REAL CONSTANT INSTEAD OF INTEGER - ROUNDING OCCURS
sys  :  ELEMENT NUMBER 12
sys  :  W LINE ENDS WITHOUT TERMINATION
sys  :  ENTER TERMINATOR OR RECOMMENCE WITH ELEMENT 14
user :  12,,,,9;
sys  :  END OF READ.  LAST DATA WAS FOR ELEMENT 18
  .
  .
  .
 ETC.
```

An existing, successfully translated program CUBICS is run, and prompting and warning messages given.

QUIT COMMAND

1. QUIT.

2. This command allows the user to terminate his NUTS session and return to Michigan Terminal System.

3. On terminal and in batch. Authors and students.

4. QUIT.

5. When the system signs the user off, it tells him how long in hours he has been signed on to NUTS to date and also how long in minutes the last session was.

6. Example.

```
user : QUIT
sys  : TOTAL TIME = 10.42 HRS.  THIS SESSION = 61.86 MINS.
sys  : # (MTS command prompt).
```

REL COMMAND

1. REL.

2. This command is used to release lessons for general use. In other words, the author wishes to allow students to use the released lesson.

3. On terminal only. Authors only.

4. REL, lname

   4.1 lname is the valid name of a lesson owned (,translated and successfully checked out) by the author issuing the command.

5. Once the system has decided that the lesson to be released is valid, it makes the further check that if this is not lesson 0, then lesson 0 must already be released. This is because lesson 0 is the fundamental lesson in a course and must be released before all others.

   It may happen that some other author has already released a lesson in a course of the same name as the course, the lesson of which is now being released. If so, the current author must first copy his lesson to one of a different name and re-translate it before trying to release it again, as identical course names between authors is not allowed.

6. Examples.

```
user : REL,MATHS0
sys  : *
```

The fundamental lesson 0 of course MATHS is released, allowing students to commence the course.

```
user : REL,HIST9
sys  : *
```

Lesson 9 of course HIST is released, assuming lesson HIST0 has been released, thus allowing students to carry on further with the course.

RES COMMAND

1. RES.

2. This command is used to withdraw lessons from general use. In other words, the author wishes to amend the lesson.

3. On terminal only. Authors only.

4. RES, lname

   4.1 lname is the valid name of a lesson owned and already released by the author issuing the command.

5.

RFILE COMMAND

1. RFILE.

2. This command gives an author a listing of the contents of any one of the response files he possesses. He uses it whilst he is checking out the logic of his courses.

3. On terminal and in batch. Authors only.

4. RFILE, rfile.

   4.1 rfile — is the valid name of a response file owned by the author himself.

5. If the response file is a valid one then the following information is returned to the author.

5.1 current position: the current lesson number of the course and the segment number in that lesson are given, together with the corresponding address in the intermediate code. A restart would occur from this position.

5.2 route: information is given in chronological order of responses. For each response this is:

   5.2.1 lesson number.

   5.2.2 question number.

   5.2.3 response type : this may be one of A(nticipated), U(nanticipated) or N(ot answered). In addition, '+D', '+C', '+P', '+DP', etc. is added depending upon whether ?DESK, ?CALC, ?PIL or some combination was used.

   5.2.4 truth value = 0 if that particular #CA/#WA element was not used.
   = 1 if that element was contained in the response.
   =-1 if not.
   This is only included if the response type is A.

   5.2.5 time taken: from keyboard unlocking to RETURN; in seconds.

   5.2.6 actual response : the first 60 characters of the response, but only if unanticipated.

5.3 times: the total time on the course and the duration of the last session are given in minutes.

6. See Appendix F for an example of a response file listing.

RID COMMAND

1. RID.

2. This command is used to destroy files; lessons, programs and response files in the case of authors but programs only in the case of students.

3. On terminal only. Authors and students.

4. RID, name

   4.1 name is the valid name of an existing file:-
   a lesson, program or response file in the case of an author.
   a program in the case of a student.

5. When a file is got rid of, it is deleted from the system and its reference in the user's catalogue removed.

6. Examples.

```
user : RID,MATHS2
sys  : *
user : RID,CUBICS
sys  : *
user : RID,MATHS#5
sys  : *
```

One of each type of file is destroyed.

SFILE COMMAND

1. SFILE.

2. This command gives an author a listing of the contents of a response file of a student who is using one of his courses. It is used to study the student's current position and progress in the course.

3. On terminal and in batch. Authors only.

4. SFILE, csname, userid

   4.1 csname is the valid name of a course owned by the author, (Naturally, some of the lessons of this course must be released).

   4.2 userid is the valid user identification number of a student who has been added to the student index, i.e. has been joined to the system.

5. After verifying the validity of the course name, the system ascertains whether the student has been joined to the system. If not, a message indicating this is returned. Then, it decides whether the student has or has not commenced this particular course. If he has, a listing of the form mentioned in paragraph 5 of the RFILE command is given.

6. See Appendix F for an example of a response file listing.

THE AUTHOR LANGUAGE

INTRODUCTION

The author language is used to design dialogues between the machine and students. Source statements written in the author language consist of a set of statements constructed by the author from the language elements described in this section.

In a process called translation, a program called the author language translater analyses the source statements and translates them into blocks of intermediate code which will subsequently be executed by the controller. In addition, when the translater detects errors in the source program, it produces appropriate diagnostic error messages.

CODING STATEMENTS

Statements are written one per line and have a maximum length of 80 characters.

CONSTANTS.

A constant is a fixed, unvarying quantity which may be either of mode integer or real.

1. Definition of integer constant - a whole number written without a decimal point. Maximum magnitude 2147483647.

2. Definition of real constant - has one of three forms: a basic real constant, a basic real constant followed by a decimal exponent, or an integer constant followed by a decimal exponent. (A basic real constant is a string of decimal digits fewer than 8, with a decimal point). Magnitude 0 or 10^{-78} through 10^{75}.

The decimal exponent permits the expression of a real constant as the product of a basic real constant or an integer constant times 10 raised to a desired power. An exponent consists of the character "¬" followed by a signed or unsigned 1- or 2-digit integer constant.

VARIABLES

A variable is a symbolic representation of a quantity that occupies a storage area. The value specified by the name is always the current value stored in the area.

The type of a variable corresponds to the type of data the variable represents. Thus, an integer variable represents integer data, a real variable real data, etc. There are 3 types of variable, 2 of which are numeric. The name of a numeric variable is from 1 through 6 alphabetic characters. There is a predefined convention used to specify variables as integer or real as follows:

1. If the first character of the variable name is I,J,K,L,M or N the variable is integer.

2. If the first character is any other letter, the variable is real. The last type of variable is of mode logical.

3. Unlike numeric variables when the author may create and name his own, there is a fixed number, 23, of logical variables to which the author only implicitly assigns values; that is, it is the student, by his reponses, who fixes the values of these logical variables.

3.1 There exist 20 response elements #CA0,#CA1,...,#CA9,#WA0,... #WA9 which give names to up to 20 responses for which an author may want to test in reply to his question. To each of these elements may be assigned any combination of strings and values. The author also indicates how many of each are required to give an "overall match" and also if ordering is to be taken into account. After the student has entered his response, each of the #CAs and #WAs becomes true or false (or undetermined if not used by the author), depending upon the response.

3.2 The implicit response element #UA becomes true if an unanticipated response is given.

3.3 The implicit response element #NA becomes true if the question was not answered; that is, the student simply hit RETURN. In that case #UA becomes false.

3.4 The implicit response element #RTn becomes true if the response was made in less than or equal to n seconds.

ARRAYS

An array is a set of variables identified by a single variable name. A particular variable in the array may be referred to by its position in the array; e.g. first variable, third variable, twentieth variable. Each variable (element) in an array consists of the name of the array immediately followed by a number enclosed in parentheses, called a subscript quantity. The variables which the array comprises are called subscripted variables. To refer to any element in an array, the array name must be subscripted. In particular, array name alone does not represent the first element.

The size and type of an array: the number of elements of an array in the author language is fixed at 100. Also, the number of subscript quantities allowed is one, making a one-dimensional array of 100 elements. An array name consists of the character "&" followed by from 1 through 6 alphabetic characters. The type of an array name is determined by the same conventions as we used to specify the type of a variable name. Each element of an array is of the type specified by the array name.

SUBSCRIPTS

A subscript is an integer subscript quantity that is used to identify a particular element of an array. It is enclosed in parentheses and written immediately after the array name. The following rules apply to the construction of subscript quantities (see sections STANDARD FUNCTIONS and EXPRESSIONS for additional information about the terms used below).

1. subscript quantities may contain arithmetic expressions using any of the arithmetic operators: +, -, *, /, %, **.

2. subscript quantities may contain standard function references.

3. subscript quantities may contain subscripted names.

4. the total nesting of subscripted names and standard function references within subscript quantities must not exceed 5.

5. mixed mode expressions (real and integer only) within subscript quantities are evaluated at run time. If the evaluated expression is real, rounding occurs.

6. the evaluated result of a subscript quantity should always be greater than zero and less than or equal to 100.

examples
```
&ARRAY(NUM)
&MATRIX(&A(1)+&E(X+#SIN(THETA)))
&X(M%3+7.2)
&A(#SIN(X+#COS(&NUM(&E(1))))) - (maximum of 5 nestings)
```

invalid
```
&ARRAY(-2)      (must not be negative)
&MATRIX(123)    (must not exceed 100)
&X(Y<GT>Z)      (must not assume a logical quantity)
```

STANDARD FUNCTIONS

The scope and value of expressions in the author language are enhanced by a facility for inserting functions, just as variables may be inserted. The name of the function, together with the appropriate argument, is merely written. The "standard" functions are some of the more frequently occurring functions of analysis and are recognised by the character "#" followed by a 3-letter mnemonic code. The argument is enclosed in parentheses and written immediately after the standard function name.

- #ABS - absolute value
- #SQT - square root, argument ≥ 0
- #SIN - sine, argument in radians
- #COS - cosine, argument in radians
- #TAN - tangent, argument in radians
- #EXP - exponential function
- #ENT - largest integer not greater than the value of the argument.
- #NLN - natural logarithm
- #CLN - common logarithm
- #ACT - arc tangent
- #ACS - arc sine
- #ACC - arc cosine

The following rules apply to the construction of arguments (see sections SUBSCRIPTS and EXPRESSIONS for additional information about the terms used below).

1. arguments may contain arithmetic expressions using any of the arithmetic operators: +, -, *, /, %, **.

2. arguments may contain standard function references.

3. arguments may contain subscripted names.

4. the total nesting of subscript names and standard function references within arguments must not exceed 5.

examples #SIN(&A(X+#COS(&NUM(&B(1))))) (maximum of 5 nestings)
#ENT(X/5+3.4)
#COS(&A(1)+&B(X+&MATRIX(M)))

invalid #SQT(Y<GT>Z) (must not assume logical quantity)

EXPRESSIONS

The author language provides three kinds of expression: arithmetic, logical and response. The value of an arithmetic expression is always a number whose type is integer or real. The value of either a logical expression or a response expression is always a truth value: true or false. Expressions may appear in assignment statements, in certain control statements and in output statements.

1. Arithmetic expressions

The simplest arithmetic expression consists of a primary which may be a single constant, variable, subscripted variable, standard function, or another expression enclosed in parentheses. The primary may be either integer or real.

1.1 Arithmetic operators

1.1.1 ** is the sign of exponentiation. The base precedes the sign and the exponent follows. The operation is effected as in ordinary arithmetic with the following comments and exceptions. No values of base and exponent which would lead to infinite, indeterminate or imaginary results are allowed, and when the exponent is real the value of the base may never be negative.

1.1.2 * multiplication, conventional meaning.

1.1.3 / real division. The operands may be of any combination but a real result occurs: e.g. 14/5 = 2.8

1.1.4 % integer division. The operands may be of any combination but first they are rounded to integers if real, before integer division yields a result as follows:

x%y = sign (rounded x/rounded y)* whole number part (modulus (rounded x/rounded y))

e.g. 14.4 % 4.6 = 14 % 5 = 2.

1.1.5 + addition, conventional meaning.

1.1.6 - subtraction, conventional meaning.

1.2 Rules for constructing arithmetic expressions

1.2.1 All desired computations must be specified explicitly; i.e. if more than one primary appears in an arithmetic expression, they must be separated from one another by an arithmetic operator, e.g. AB is written A*B if multiplication is intended.

1.2.2 No two arithmetic operators may appear in sequence in the same expression.

e.g. A*/B and A*-B. To correct the latter, parentheses are added, A*(-B).

1.2.3 Order of computation: computation is performed from left to right according to the hierarchy of operations thus:

Hierarchy	Operation
first (highest)	evaluation of standard functions
second	exponentiation (**)
third	multiplication and division(*, / and %)
fourth	negation (-)
fifth	addition and subtraction (+ and -)

This hierarchy is used to determine which of two consecutive operations is performed first. If the first operator is higher than or equal to the second, the first operation is performed. If not, the second operator is compared with the third, etc. When the end of the expression is encountered, all of the remaining operations are performed in reverse order.

Parentheses may be used in arithmetic expressions, as in algebra, to specify the order in which the arithmetic operations are to be computed. Where parentheses are used, the expression within the parentheses is evaluated before the result is used This is equivalent to the definition above since a parenthesised expression is a primary.

2. Logical expressions

The simplest form of logical expression consists of one of the 23 pre-named logical variables, or a logical expression enclosed in parentheses, which always has the value true or false.

More complicated logical expressions may be formed by using logical and relational operators. These expressions may be in one of the following three forms:

relational operators combined with arithmetic expressions OR
logical operators combined with logical primaries OR
logical operators combined with either or both forms of the logical expressions described in the first two forms.

2.1 Relational operators

The 6 relational operators, each of which must be preceded by "<" and followed by ">" are as follows:

<GT> greater than (>)
<GE> greater than or equal to (≥)
<LT> less than (<)
<LE> less than or equal to (≤)
<EQ> equal to (=)
<NE> not equal to (≠)

The relational operators express an arithmetic condition which can be either true or false. Only arithmetic expressions may be combined by relational operators.

examples A<GT>N
X **3.1<EQ>(Z+W)*3
.6<LE>EPS

invalid #CA3<EQ>X+Y (logical quantity not allowed)
X **3<LT3.2 (">" missing)
<GT>7 (missing arithmetic expression)

2.2 Logical operators

The 3 logical operators, each of which must be preceded by "<" and followed by ">" are as follows. Each takes its conventional meaning.

Two logical operators may appear in sequence only if the second one is the logical operator <NOT>. Only those expressions which, when evaluated, have the value true or false may be combined with the logical operators to form logical expressions.

examples &ARRAY(X+2) <GT> 0 <AND>#WA9
#CA2 <AND><NOT>#WAO

invalid X <AND>#CA2 (X not a logical expression)
<OR> X <GT> Y (<OR> must be preceded by a logical expression)
#CA2 <AND><OR>#CA3 (<AND> and <OR> must be separated by a logical expression)
#CA2 <NOT><AND>#CA3 (<NOT> must not precede <AND> or <OR>)

2.3 Order of computation in logical expressions

The order in which the operations are performed is:

Hierarchy	Operation
first (highest)	evaluation of standard functions
second	exponentiation (**)
third	multiplication and division (*, / and %)
fourth	negation (-)
fifth	addition and subtraction (+ and -)
sixth	<GT>, <GE>, <LT>, <LE>, <EQ>, <NE>
seventh	<NOT>
eighth	<AND>
ninth	<OR>

Parentheses may be used in logical expressions to specify the order in which the operations are to be performed. When parentheses are used, the expression contained within the most deeply nested parentheses (i.e. the innermost pair of parentheses) is effectively evaluated first.

The logical expression to which the logical operator <NOT> applies must be enclosed in parentheses if it contains two or more quantities. e.g., if A and B represent logical primaries, then the following two expressions are not equivalent.

<NOT> (A <OR> B) ; <NOT> A <OR> B

In the first <OR> operates on A and B first then <NOT> acts on the result. In the second <NOT> acts on A first and then <OR> operates on this result and B.

3. Response expressions

A complete description of response expressions appears in the section ASSIGNMENT STATEMENTS.

THE PAST STUDENT PERFORMANCE FACILITY, #PERF.

This is a logical function which acts in some ways like a standard function in that it is invoked merely by writing it together with an

appropriate argument contained in parentheses. The difference is that the one result that is returned may only have the value true or false.

There are 8 kinds of past performance about which the author may inquire, all concerned with the student's record during the current lesson. In all types, the range of question number is 1 through 99 and that of answer number from 0 through 9.

1. qCAd - student matched Correct Answer d to question q. e.g. #PERF (26CA3)
2. qWAd - student matched Wrong Answer d to question q. e.g. #PERF (3WA0)
3. qNS - student has Not Seen question q yet. e.g. #PERF (99NS)
4. qNA - student did Not Answer question q. e.g. #PERF (17NA)
5. qUA - student gave an Unanticipated Answer to question q. e.g. #PERF (36UA)
6. qRTn - student answered question q in less than or equal to n seconds. e.g. #PERF (46RT10)
7. (q1,q2, ...,q3-q4,...)nXX where n is an unsigned integer and XX may be any one of CA, WA, NS, NA or UA with the appropriate meaning.

   - out of question q1, q2, ...,q3,q3+1,q3+2, ...,q4, ... the student satisfied the XX property n times, e.g. #PERF (7,8,11-15, 19,23-26)10CA. This means that out of the 12 specified questions, the student gave 10 correct answers, no matter which answer number.

8. q1-q2-q3-q4- .... - the student's path through the lesson was successively questions q1, q2, q3, q4 ... . e.g. #PERF (7-8-11-12-12-13). This means that the student's route through the questions was from 7 to 8 to 11 to 12 (two attempts) to 13.

To supply a more complex argument to #PERF, it is possible to test the conjunction of combinations of each of these 8 kinds of past performance by simply separating each different inquiry by a comma. Naturally, if one enquiry proves false, the whole #PERF call gives back the value false.

Example #PERF (1CA4,2WA1,3NS,4NA,5UA,6RT30, (7,8,11-15)5CA,16-19-20)

this is true if

the student's response to question 1 contained correct answer 4 AND
his response to question 2 contained wrong answer 1 AND
he has not seen question 3 yet AND
he did not answer question 4 AND
he gave an unanticipated answer to question 5 AND
he answered question 6 within 30 seconds AND
he made at least 5 correct responses (of any answer number) to questions 7,8,11,12,13,14,15 AND
he proceeded from question 16 to 19 to 20.

ASSIGNMENT STATEMENTS

There are two types of assignment statements: arithmetic and response.

1. Arithmetic assignment statement

1.1 non-subscripted variable

a = b , where a is a non-subscripted variable and b is an arithmetic expression. This statement closely resembles a conventional algebraic equation; however, the equal sign specifies replacement rather than equivalence; i.e. the expression to the right of the equal sign is

evaluated, and the resulting value replaces the current value of the variable to the left of the equal sign. The arithmetic expression must not contain any variable which has not been previously assigned a value.

Example DIST = TIME*VEL

1.2 array

a = b1,b2,b3, ... bN , where a is a subscripted variable or an array name only and b1,b2,b3,..., bN are arithmetic expressions.

If the array name only is given, the subscript is assumed to be 1, the first element of the array. The expressions to the right of the equal sign are evaluated, and the resulting values replace the current values of the successive array elements commencing with that specified on the left of the equal sign. The arithmetic expressions of the subscript quantity and the right hand side must not contain any variable which has not been previously assigned a value.

Example &ARRAY(12) = 0,0,X,Y,3,0

After this statement, values of array &ARRAY become:

&ARRAY (1) to &ARRAY (11) unchanged
&ARRAY (12) = 0
&ARRAY (13) = 0
&ARRAY (14) = value of X
&ARRAY (15) = value of Y
&ARRAY (16) = 3
&ARRAY (17) = 0
&ARRAY (18) = to &ARRAY (100) unchanged.

2. Response assignment statement

$$r=\left['s1'\left[\left\{\begin{matrix}i1\\K\end{matrix}\right\}\right],'s2'\left[\left\{\begin{matrix}i2\\K\end{matrix}\right\}\right],\ldots.\right],\left[<v1\left[,e1\right]>,<v2\left[,e2\right]>,\ldots\right],\left[Sm\left[\left\{\begin{matrix}O\\U\end{matrix}\right\}\right]\right],\left[Vn\left[\left\{\begin{matrix}O\\U\end{matrix}\right\}\right]\right.$$

where r is a response element,

s1,s2, ... are strings of characters,
i1,i2, ... are unsigned integers,
v1,v2, ... are "exact value" arithmetic expressions,
e1,e2, ... are "error" arithmetic expressions,
and m,n are unsigned integers.

The strings, values, string number specification and value number specification may be in any order so long as they are separated by commas.

At least one string or value must be present.

After asking the student a question and before indicating when the response is required, the author MUST specify at least one expected response. He does this by making a response assignment. There exist 20 response elements, #CA0, #CA1, ..., #CA9, #WA0, ..., #WA9, which give names to up to 20 expected responses for which an author may want to test.

2.1 To each of these elements may be assigned any combination of strings and values.

2.1.1 A string is simply a collection of characters enclosed by " ' "s and optionally followed by either an unsigned integer or the letter "K".

2.1.1.1 If neither unsigned integer nor "K" follows the string, then the author requires the exact string in the student's response.

e.g. 'FORTRAN' - requires exactly "FORTRAN" somewhere in the response.

2.1.1.2 If an unsigned integer follows the string, this indicates the maximum number of characters which may be wrong in a string of the same length from the student's response and yet still provide a match.

e.g. 'FORTRAN'1 - requires "*ORTRAN","F*RTRAN", FO*TRAN", etc., where * is any character, somewhere in the response. However, "FOURTRAN" does not match as the system is searching for a 7-character string with only one error. 7 consecutive characters from "FOURTRAN" has at minimum 2 errors. ("OURTRAN").

2.1.1.3 If the letter "K" follows the string, a Kernel match is sought. This is the specification of certain characters (no blanks) from a required answer and a match will occur if the student's response contains a string of any length which has the characters in the given order.

e.g. 'FTRN'K - requires "FTRN", "FORTRAN", "FOURTRAN", "FARTRON", etc. somewhere in the response.

2.1.2 A value is any arithmetic expression enclosed by "<" and ">". However, if the value is to have error bounds, that is, an exact value match is not required, then there are two arithmetic expressions separated by a comma and all enclosed by "<" and ">". In this case, the first expression indicates the value sought and the second expression the allowable error.

Examples <1066> - the integer 1066 in the student's response gives a match.
<1050,50> - any integer from 1000 through 1100 gives a match.

2.2 Any combination of strings and values, each separated by a comma, may be assigned to a response element. Consequently, the number of strings, whether they must be ordered or not, the number of values, whether they must be ordered or not, all must be specified to indicate what the author requires to be an "overall match" by the student's response. This the author does by adding, somewhere within the assignment, after a comma (unless immediately after "="), a specification for strings followed by a comma (unless at the end of the line) and, after a comma (unless immediately after "="), a specification for values followed by a comma (unless at the end of the line). These take the form

$$\begin{Bmatrix} S \\ V \end{Bmatrix} n \left[\begin{Bmatrix} O \\ U \end{Bmatrix} \right]$$

where S/V stands for strings/values,

n is an unsigned integer giving the number required

and O/U stands for ordered/unordered.

O/U is optional and defaults to O if absent. Both these specifications are themselves optional and default to one only of that type for an "overall match".

Once the student has entered his response, the response elements become true or false (or undetermined if not used by the author) depending upon the response.

Example #CA3 = 'computer', 'ftrn'K, S20, <1970>, <1969>

#CA3 will become true if the student's response contains both "computer" (exactly) and a string containing "ftrn" in that order, in that order and either integer 1969 or integer 1970; that is, the response "In 1969 our computer will use more fourtran than at present".

LABELS

There are three different types of label:

1. Statement labels. Any statement except END may be labelled for reference from other statements. The statement label consists of from 1 through 3 decimal digits (except that the label value 0 is not permitted) followed by a right parenthesis. The statement follows this. These statement labels may be assigned in any order and their value does not affect the order in which the statements are executed in the lesson.

Examples	9)A=1	invalid	00)A=1 (value 0 not permitted)
	983)STOP		2137)X=2 (only up to 3 digits permitted)
			27)END (statement END must not be labelled).

2. Segment labels. The author usually divides his lesson into segments as the system restarts a student in the segment he was in when he terminated his last session with the course. Any statement except END may be labelled as the start of a segment. In order to use the segment label as a restart point, the system stores all variable and stack values when it encounters the segment label during execution of the course. Also, it cancels the current addresses contained in the 4 control address registers (see statement CTRL). The segment label consists of the letter "S" followed by a 1- or 2- digit number (except that label value 0 is not permitted) followed by a right parenthesis. The statement follows this. These segment labels may be assigned in any order and their value does not affect the order in which the segments are executed in the lesson.

Examples	S3)A=1	invalid	S0)A=3 (value 0 not permitted)
	S75)X=3		S225)E=1 (maximum of 2 digits permitted)
			S31)END (statement END must not be labelled)

3. Question labels. Whenever the author wishes to ask the student to make a response, he inserts a question label on the statement which commences the question. This ensures that during execution of the course, the system will store the student response information in the correctly indexed position in the response file. Any statement except END may be labelled as the start of a question. The only

restriction is that there exists at least one response assignment between the question label and the response request. The question label consists of the letter "Q" followed by either a 1- or 2-digit number (except that label value 0 is not permitted) followed by a right parenthesis. The statement follows this. These question labels may be assigned in any order and their value does not affect the order in which the questions are executed in the lesson. Reference to question numbers in the #PERF facility is to the 1- or 2- digit number following the "Q".

Examples	Q5)A=1	invalid	Q00)A=5 (value 0 not permitted)
	Q83)X=2		Q123)NUM=5 (only maximum of 2 digits permitted)
			Q52)END (statement END must not be labelled)

Source statements may be multi-labelled with different kinds of label but there exists a priority of:

segment label before question label before statement label.

example S27)Q43)297)A=3

invalid Q5)S17)B=2 (segment label must precede question label)

Q7)103)21)N=N+1 (multi-labelling must not contain 2 or more labels of the same type).

CONTROL STATEMENTS

1. JUMP STATEMENT

JUMP> label

where label is either a statement label, segment label or question label. JUMP statements permit transfer of control to another executable statement specified by a label of any type. Used on its own, the JUMP statement causes unconditional transfer, but, together with an IF statement, the transfer is conditional.

examples	JUMP> 157	invalid	JUMP83 (">" is missing)
	JUMP> S73		
	JUMP> Q2		

2. LOAD STATEMENTS

LOADn>label

where n is a digit from 1 through 5.
and label is either a statement label, segment label or question label.

There exist 5 stacks each of 10 elements which the author may use to stack return addresses (specified by labels) and subsequently unstack when returning control to these addresses via the RETN statements. The 5 stacks are known by the digits 1 to 5 and are referenced by virtue of supplying the required digit to the statement itself.

examples	LOAD1>853	invalid	LOAD2S71 (">" is missing)
	LOAD3>S17		LOAD>Q3 (digit specifying the stack is missing)
	LOAD4>Q2		LOAD6>91 (only stacks 1 through 5 exist)

3. RETN STATEMENTS

RETNn

where n is a digit from 1 through 5.

This statement looks at the return address on the top of stack n, unstacks, then returns control to this address. If the particular stack specified is empty when RETNn is encountered, the statement is ignored and control passes to the next statement.

examples RETN1
RETN4

invalid RETN (digit specifying stack absent)
RETN6 (stack 6 does not exist)

4. TRANS STATEMENT

TRANS (lesson number, segment number)

where lesson number is a valid lesson number from 1 through 9 and segment number is a valid segment number from 1 through 99.

As courses may consist of up to ten lessons, the author must transfer control out of lesson 0, the fundamental lesson, at some stage to another lesson if more than one is used. If more than two are used, of course, then control may be transferred from second to third as well as back to first, etc. To effect this, the author uses the TRANS statement in which he specifies the lesson to which he wishes to transfer control and also the particular segment which will be the continuation point in that lesson. A TRANS statement indicates a dynamic end of the lesson.

examples TRANS (0,72) - transfer control to segment 72 of lesson 0.
TRANS (8,19) - transfer control to segment 19 of lesson 8.

invalid TRANS (12,61) - invalid lesson number.
TRANS (3,192) - invalid segment number.

Of course, a segment number may be valid but not exist in the lesson specified. In that case, an error message is returned at run time.

5. CTRL STATEMENT

CTRL (label1, label2, label3, label4)

where label1, label2, label3 and label4 are each either a statement label, segment label or question label.

As explained in the COURSE COMMAND, the student may enter predetermined responses when the keyboard is unlocked to him. These are non-solution responses and have other functions. Four of these are "?F", "?B", "?S" and "?G". There exist 4 control address registers which, at the beginning of a segment, are emptied. However, the author may supply four valid addresses to these registers by using a CTRL statement to supply four labels of any type and combination. Then, after this CTRL statement but before the start of another segment or, indeed, another CTRL statement in the same segment, if the student enters "?F", "?B", "?S" or "?G", control is passed to the corresponding address given by the first, second, third or fourth

parameter, respectively, of the CTRL statement. If the author does not supply a CTRL statement within a segment or if he asks the student to make a response before the CTRL statement is effective in a segment, entry of one of the four predetermined responses by the student is ignored as such. The response is taken as either an actual response to a question or a request to end a pause, depending on the circumstance.

example CTRL (S19, Q33, S7, 237)

This statement means that up to the next CTRL statement or segment start, whichever is the sooner:

entry of	"?F"	causes control to pass to		S19
"	"	"?B"	"	Q33
"	"	"?S"	"	S7
"	"	"?G"	"	237

6. IF STATEMENT

IF (logical expression) statement

where logical expression is any valid logical expression and statement is any statement except END or another IF statement.

The IF statement is used to evaluate the logical expression contained in parentheses and to execute or skip the statement depending on whether the value of the expression is true or false, respectively.

example IF(#CA3<AND>N<LT>3)JUMP>Q15

This statement specifies that control is to be transferred to question 15 if the student's response to the last question contained the author's correct answer 3 and the value of the variable N is less than 3. Otherwise, the next statement is executed.

7. PAUSE STATEMENT

PAUSE

The PAUSE statement causes the keyboard to unlock after printing ":" on a new line. This character indicates to the student that he may pause at this point in the course if he desires, without the system displaying any more text or requesting any more responses. To continue, the student simply presses RETURN.

8. STOP STATEMENT

STOP

The STOP statement terminates the execution of the course. It thus indicates a dynamic end of the lesson.

9. END STATEMENT

END

The END statement is a non-executable statement that defines the static end of a lesson for the translater. Physically, it must be the last statement of the lesson. It may not be labelled, nor be

the statement part of an IF statement. The END statement does not terminate execution of the lesson. To terminate execution of the lesson, a STOP or TRANS statement is required. Thus, each lesson must contain at least one of either of these.

INPUT/OUTPUT STATEMENTS

1. RESP STATEMENT

$$\text{RESP}\left[\langle v1\rangle\langle v2\rangle\ldots\langle s1\rangle\langle s2\rangle\ldots\langle a1,\begin{Bmatrix}w1\\c1\end{Bmatrix}\rangle\langle a2,\begin{Bmatrix}w2\\c2\end{Bmatrix}\rangle\ldots\right]$$

where v1, v2 ... are non-subscripted variables,
s1, s2 ... are subscripted variables,
a1, a2 ... are array names,
w1, w2 ... are non-subscripted variables,
and c1, c2 ... are constants.

The RESP statement unlocks the keyboard for a student's response. It should only be entered if the author has made at least one response assignment since he inserted a question label. The statement may just be the statement name itself. However, if the author desires to store any numerical values contained in the student's response, he specifies variables into which the successive values will be placed. He may specify these variables in any of three ways and use any combination of these three ways, in any order:

1.1 non-subscripted variable - the variable name is enclosed in "<" and ">", e.g. <LENGTH>.

1.2 subscripted variable - the array name followed by a subscript quantity within parentheses, all enclosed in "<" and ">", e.g. <&ARRAY(I+2)>.

1.3 array name with starting point - the array name followed by a comma then either a constant or a non-subscripted variable, all enclosed in "<" and ">". The significance of the constant or non-subscripted variable is to denote the element of the array where the next in the series of values is to be placed and thereafter the values will be placed in successive elements, e.g. <&ARRAY,START> or <&MATRIX,91>.

If there are more variables specified than numerical values available from the response, the variables not yet assigned a value are given the value 999 999 999.

example RESP<NUM><&FIRST(5)><&MATRIX,96><&ARRAY,96>

if the student's reply was:

"There have been 10 leap years: 1932, 1936, 1940, 1944, 1948, 1952, 1956, 1960, 1964 and 1968."

then the above variables will be assigned as follows:

NUM=10
&FIRST(5)=1932
&MATRIX(1) TO &MATRIX(95) - unchanged
&MATRIX(96) TO &MATRIX (100)=1936, 1940, 1944, 1948, 1952, respectively.
&ARRAY(1) TO &ARRAY(95) - unchanged
&ARRAY(96) TO &ARRAY(100)= 1956, 1960, 1964, 1968, 999 999 999, respectively.

2. TYPE STATEMENT

TYPE ('str1'p1'str2'p2 ... <ae1>q1,w1,d1<ae2>q2,w2,d2...)

where str1, str2 ... are strings of characters,
p1, p2 ... are unsigned, non-zero integers,
ae1, ae2 ... are arithmetic expressions,
q1, q2 ... are unsigned, non-zero integers,
w1, w2 ... are unsigned, non-zero integers,
and d1, d2 ... are integers, signed or unsigned.

The TYPE statement is used to display character strings and arithmetic expression values in specified positions on the terminal/line printer.

2.1 Any character may be contained in the string within single quotes, with the restrictions that " ' " must be represented as " '' " and " ! ", "ø", " _ " and "backspace" must be represented as "!!", "!ø", "!_ " and "! backspace", respectively.

2.2 The unsigned integer after the string indicates the starting column of the string.

2.3 There may be a maximum of 10 strings within a TYPE statement.

2.4 Any valid arithmetic expression may be contained within "<" and ">" so long as each variable within the expression has previously been assigned a value.

2.5 The q's, w's and d's in the description of the TYPE statement represent the starting column of the value of the expression, the width of field of this value, and the number of decimal places to the right of the decimal point, respectively.

2.5.1 if d=0, then an integer is printed.

2.5.2 if d>0, then a fixed point number is printed. Note that w≥d+3 to allow for sign, decimal point and at least one digit to the left of the decimal point.

2.5.3 if d<0, then a floating point number is printed. Note that w≥ modulus (d)+7 to allow for sign, decimal point, at least one digit to the left of the decimal point, letter E for exponent, sign for exponent and two digits for magnitude of exponent.

2.5.4 modulus (d) must not be greater than seven, the maximum precision allowed.

2.6 There may be a maximum of 5 expressions within a TYPE statement.

2.7 There must be at least one string or expression within the parentheses of the TYPE statement.

example

```
TYPE('MEAN'1<SUM/N>6,10,5'VAR'21<SUMSQ-SUM**2/N>25,12,-5'N='41<N>43,
     2,0)
```

If the current values of SUM, N and SUMSQ are 65, 10 and 441, respectively and if "s" represents a space, then this statement will give the output:

MEANssss6.50000sssssVARss1.85000E+01ssssN=10

3. BACK STATEMENT

BACK ('str1'p1'str2'p2 ...<ae1>q1,w1,d1<ae2>q2,w2,d2 .....)

where the abbreviations and their meaning are exactly the same as for the TYPE statement.

Since the parentheses and their contents are optional there is one difference. The BACK statement returns the student to the last RESP statement for another attempt at the question. If the parentheses are absent, no message is given back before the next response, but if present, the contents are displayed first.

SAMPLE PROGRAM

See Appendix D.

THE CALCULATING LANGUAGE

INTRODUCTION: the calculating language is used in writing programs for applications that involve mathematical computations and other manipulation of numerical data. Source statements written in the calculating language consist of a set of statements constructed by the user from the language elements described in this section.

In a process called translation, a program called the calculating language translater analyses the source statements and translates them into blocks of intermediate code which will subsequently be executed by the controller. In addition, when the translater detects errors in the source program, it produces appropriate diagnostic error messages.

CODING STATEMENTS

Statements are written one per line and have a maximum length of 80 characters.

CONSTANTS

As author language.

VARIABLES

As author language.

ARRAYS

As author language.

SUBSCRIPTS

As author language.

STANDARD FUNCTIONS

As author language.

EXPRESSIONS

The calculating language provides two kinds of expression : arithmetic and logical. The arithmetic is the same as that contained in the author language.

The simplest form of logical expression consists of two arithmetic expressions separated by a relational operator, or a logical expression enclosed in parentheses, which always has the value true or false.

More complicated logical expressions may be formed by using logical operators combined with logical primaries.

ASSIGNMENT STATEMENTS

There is only one type of assignment statement : the arithmetic assignment statement. This is the same as that in the author language.

LABELS

There is only one kind of label allowed in the calculating language :

the statement label. It is similar to the statement label of the author language except that it may only consist of a 1- or 2- digit number.

CONTROL STATEMENTS

1. JUMP STATEMENT

As author language, to a statement label.

2. IF STATEMENT

As author language.

3. STOP STATEMENT

As author language.

4. END STATEMENT

As author language.

INPUT/OUTPUT STATEMENTS

1. READ STATEMENT

$$\text{READ}\left(\left\{\begin{matrix}\text{nsv1}\\ \text{sv}\\ \text{an}\left[,\ \left\{\begin{matrix}\text{nsv2}\\ \text{c}\end{matrix}\right\}\right]\end{matrix}\right\}\right)$$

where nsv1, nsv2 are non-subscripted variables,
sv is a subscripted variable,
an is an array name,
and c is a constant.

The READ statement unlocks the keyboard to allow the user to enter the data he requires. A prompt is given for every READ statement at run time, indicating either the variable name alone or the variable name and either subscript quantity or starting point, whichever is applicable. There are basically three types of READ statement:

1.1 non-subscripted variable - a valid variable name only is entered between the parentheses; e.g. READ(LENGTH), READ(DIST).

1.2 subscripted variable - a valid array name followed by a subscript quantity within parentheses are all entered between further parentheses;
e.g. READ(&ARRAY(X+#SQT(Y))), READ(&MATRIX(I))

1.3 array name with starting point - the array name optionally followed by a comma then either a constant or a non-subscripted variable, all enclosed in parentheses. The significance of the constant or non-subscripted variable is to denote the element of the array where the first of the series of constants entered after prompt at run time is to be placed, and, thereafter, the constants will be placed in successive elements. If the comma and the constant or non-subscripted variable are omitted, then the array element to be assigned the first input constant defaults to be the first element of

the array.
E.g. READ(&ARRAY,START), READ(&MATRIX,59), READ(&NUM)

For description of prompts and use of READ at run time, see PROG COMMAND.

2. TYPE STATEMENT

As author language

SIMPLE PROGRAMS

The first program finds all the prime numbers up to 100 by the Sieve of Eratosthenes Method. The second program reads in a series of ages 0 to 99 followed by a terminator greater than 99, then groups the ages into 0-19, 20-39, etc.

1. Prime numbers

```
TYPE('FOLLOWING IS A LIST OF PRIME NUMBERS FROM 2 TO 100'1)
TYPE('2'5)
TYPE('3'5)
I = 5
1)K =#SQT(I)-0.5
J = 3
2)IF(I%J*J<EQ>I)JUMP>3
J = J+2
IF(J<LE>K)JUMP>2
TYPE(<I>1,5,0)
3)I = I+2
IF(I<LE>100)JUMP>1
TYPE('END OF LIST'1)
STOP
END
```

2. Age grouping

```
TYPE('ENTER THE ELEMENT WHERE THE FIRST AGE IS TO BE STORED'1)
READ(I)
TYPE('ENTER A SERIES OF AGES BETWEEN 0 AND 99 TERMINATED BY A
     NUMBER'1)
TYPE('GREATER THAN 99'1)
READ(&IAGE,I)
J = 1
1)&M(J)=0
J = J+1
IF(J<LE>5)JUMP>1
2)IF(&IAGE(I)<GT>99)JUMP>3
&M(&IAGE(I)%20+1)=&M(&IAGE(I)%20+1)+1
I+I+1
JUMP>2
3)TYPE('AGES GROUP AS FOLLOWS:'1)
TYPE('0-19'3'20-39'13'40-59'23  '60-79'33'80-99'43)
TYPE(<&M(1)>1,5,0<&M(2)>11,5,0<&M(3)>21,5,0<&M(4)>31,5,0<&M(5)>
     41,5,0)
STOP
END
```

APPENDIX B.

The post-test.

1. All the following PIL statements contain one error. In each case say what it is.

   (a) SET b=c*d-e — 2 marks

   (b) x=sqrt of (a*b). — 2

   (c) SETy=x**2-2*x+1. — 2

   (d) IF x>y TYPE x. — 3

   (e) FOR I=1 TO N SET A(I)=I. — 3

2. Given the following sequence of instructions:

   SET A=3.

   SET B=1.

   SET X(A,1)=1.

   SET X(2,B)=2.

   SET X(B+A-3, (A+B)/2)=3.

   SET C=A**2-4* B-A+1 .

   What are the values of the following expressions?

   (a) C — 2 marks

   (b) cos of 3.14159*2 — 3

   (c) X(1,2)**X(2,1) — 2

   (d) X(1,3) — 2

   (e) ip of ((B**2-4*A*C)/2/A — 3

   (f) A ¢GT B ¢AND ¢NOT C=0 — 2

3. Look at the following sequence:

   =1.1 demand a,b,c.

   =do part 1.

   a=

   =4.

   b=

   =a+sqrt of a.

   c=

   =a*b.

Is this valid?

If yes, a= ,b= ,c= 6 marks

4. Write a statement that will:

print a number m if it is divisible by 7.

(Hint: use "ip of (m/7)*7") 14 marks

5. Write a statement that will:

do nothing if x is negative or greater than 1000,

otherwise set y equal to x. 14 marks

6. After the sequence:

=1.1 set s=0.

=1.2 for j=1 to 20 by j : do part 2.

=2.1 set s=s+j.

=2.2 set j=j+1.

=do part 1.

what are the final values of s and j?

s= ,j= 16 marks

7. Given that

s1= "abc"

s2= "01234"

s3= ""

s4= "bcd012"

what are the values of the following expressions:

(a) s1+s2 3 marks

(b) s2<s4 3

(c) 1 of (s2+s3+s4) 3

(d) 2 ØFC s4 3

(e) 3 ØLC s4+"34"+s3 = 5 ØFC s2 3

(f) 2 ØFC 4 ØLC s4 3

(g) THE VALUE OF s2 3

(h) s3=" " 3

APPENDIX C.

The attitude questionnaires.

The pre-questionnaire.

1. Name.

2. Have you had any previous programming experience? YES/NO.

3. If yes, how much? DAYS/WEEKS/MONTHS. What languages?

4. Do you have any objections to being taught by means of a typewriter terminal? YES/NO. If yes, please state them.

(For the remaining questions, the student was asked to choose one of VERY TRUE/TRUE/FALSE/VERY FALSE. The results are given in this order for each question.)

		A	B	C	total
6.	I would prefer to proceed through the course at my own pace rather than at the lecturer's pace.	2 4 1 0	1 4 1 0	3 4 0 0	7 13 2 0
7.	Being asked questions on the subject matter from time to time is a good idea.	1 6 0 0	0 6 0 0	5 2 0 0	6 16 0 0
8.	It is a decided nuisance having to wait for the course material to be typed out in front of me.	0 2 5 0	0 0 6 0	0 1 5 1	1 3 17 1
9.	Listening to a typewriter for an hour or more typing away would do my nerves no good.	0 1 6 0	0 1 3 2	0 2 4 1	0 4 15 3
10.	When attending a course, I like to receive printed lecture notes (i.e. hard copy).	6 1 0 0	4 2 0 0	1 4 2 0	12 8 2 0
11.	Any method of teaching is better than lecturing.	0 1 4 2	0 2 3 1	0 0 5 2	1 3 13 5
12.	In a practical class, I prefer demonstrators to approach me periodically rather than have to ask them for help.	1 5 1 0	1 2 3 0	0 6 1 0	2 14 6 0

		A	B	C	total
13.	I learn more from practical classes than from lectures.	3 3 1 0	0 4 2 0	4 6 1 0	4 14 4 0
14.	I prefer a lecturer to give me good notes on the blackboard rather than an inspiring lecture.	2 5 0 0	1 4 1 0	2 4 1 0	6 13 3 0
15.	When conversing with the typewriter, I prefer to be told if I make a wrong answer.	2 5 0 0	4 2 0 0	1 6 0 0	8 14 0 0
16.	I prefer to be given my overall score on questions from time to time.	1 6 0 0	2 4 0 0	0 6 1 0	3 18 1 0
17.	My typing is too slow for this method of teaching.	1 4 2 0	2 4 0 0	1 5 1 0	4 15 3 0
18.	I would prefer a noiseless, swift, visual display to a typewriter, even though I would not get a hard copy.	0 1 6 0	0 0 5 1	1 1 5 0	2 2 17 1
19.	This method of teaching would be better if two people sat at one typewriter.	1 3 3 0	0 4 1 1	0 3 3 1	2 10 8 2
20.	I would prefer to be taught in a definite sequence of topics rather than choose for myself the topics I learn and when I learn them.	2 4 1 0	0 4 2 0	1 4 2 0	4 13 5 0

The post questionnaire.

	For group B and group C:		B	C
1.	The "pauses" allowed me to proceed at my own pace.	TRUE	6	6
		FALSE	0	1
2.	The number of pauses was	TOO LARGE	0	0
		TOO SMALL	3	5
		ABOUT RIGHT	3	2

			B	C
3.	The number of questions I was asked during the lessons was	TOO FEW	1	0
		TOO MANY	0	0
		ABOUT RIGHT	5	7
4.	The questions were	TOO EASY	0	0
		TOO DIFFICULT	0	0
		ABOUT RIGHT	6	7
5.	I prefer multiple choice questions to those where I had to enter my own answer.	VERY TRUE	0	0
		TRUE	2	0
		FALSE	2	6
		VERY FALSE	2	1
6.	In some questions I did not gather what was required.	VERY TRUE	1	0
		TRUE	4	5
		FALSE	1	2
		VERY FALSE	0	0
7.	It is a decided nuisance having to wait for the course material to be typed out in front of me.	VERY TRUE	0	1
		TRUE	4	0
		FALSE	1	5
		VERY FALSE	1	1
8.	Listening to a typewriter for an hour or so got on my nerves.	VERY TRUE	0	0
		TRUE	0	0
		FALSE	3	4
		VERY FALSE	3	3
9.	The notes I received from the typewriter were	VERY GOOD	1	3
		GOOD	4	4
		POOR	1	0
		VERY POOR	0	0
10.	This method of teaching is preferable to lectures. (N.B. 1 abstension).	VERY TRUE	1	1
		TRUE	4	2
		FALSE	1	3
		VERY FALSE	0	0
11.	I learned more from the practical sessions than from the lessons. (N.B. 1 abstension).	VERY TRUE	0	0
		TRUE	3	2
		FALSE	3	4
		VERY FALSE	0	0
12.	This method of teaching is too impersonal.	VERY TRUE	0	0
		TRUE	3	4
		FALSE	3	3
		VERY FALSE	0	0
13.	However, if reinforced by small tutorial classes, this would not be the case.	VERY TRUE	2	1
		TRUE	3	5
		FALSE	1	1
		VERY FALSE	0	0
14.	The number of feedback messages to my responses was:	TOO FEW	2	1
		TOO MANY	1	2
		JUST RIGHT	3	4

			B	C
15.	I prefer the feedback messages to be chatty, not just "YES" or "NO".	VERY TRUE	1	0
		TRUE	3	5
		FALSE	1	0
		VERY FALSE	1	2
16.	I was told my overall score on series of questions:	TOO OFTEN	0	2
		TOO INFREQUENTLY	2	2
		ABOUT THE RIGHT FREQUENCY	4	3
17.	My typing is too slow for this method.	VERY TRUE	0	0
		TRUE	2	3
		FALSE	4	3
		VERY FALSE	0	1
18.	I would prefer a noiseless, swift, visual display to a typewriter, even though I would not get a hard copy.	VERY TRUE	0	0
		TRUE	0	0
		FALSE	4	4
		VERY FALSE	2	3
19.	This method of teaching would be better if two people sat at one terminal.	VERY TRUE	1	0
		TRUE	0	1
		FALSE	3	5
		VERY FALSE	2	1
20.	The response from the system was	TOO SLOW	6	7
		O.K.	0	0
21.	The subject content of the lessons was	TOO MUCH	2	1
		INSUFFICIENT	3	3
		ABOUT RIGHT	1	3
22.	The number of worked examples was	TOO FEW	3	2
		TOO MANY	0	1
		ABOUT RIGHT	3	4
23.	I did not have sufficient time each day.	VERY TRUE	2	5
		TRUE	4	2
		FALSE	0	0
		VERY FALSE	0	0
24.	Three days is too short for such a course.	VERY TRUE	3	2
		TRUE	1	4
		FALSE	2	1
		VERY FALSE	0	0
25.	This method of teaching would be fine for one or two hours per week.	VERY TRUE	0	1
		TRUE	5	3
		FALSE	0	3
		VERY FALSE	1	0
26.	This method of teaching would be better for end of year revision rather than original course work.	VERY TRUE	1	1
		TRUE	2	4
		FALSE	3	2
		VERY FALSE	0	0
27.	Would you like to be able to ask the typewriter questions?	YES	6	6
		NO	0	1

Group C only

28.	Four questions in any examples class was	TOO MANY	4
		TOO FEW	0
		ABOUT RIGHT	3
29.	I prefer to be taught in a definite sequence of topics (like a lesson) rather than decide what I do next (in an examples session).	VERY TRUE	0
		TRUE	3
		FALSE	3
		VERY FALSE	1
30.	In the examples classes, I liked the idea of requesting that help which I wanted, not what the computer thought I wanted.	VERY TRUE	2
		TRUE	5
		FALSE	0
		VERY FALSE	0

Response matrices for comparison questions.

(i) It is a decided nuisance having to wait for the course material to be typed out in front of me.

	B				C				total				
	VT	T	F	VF	VT	T	F	VF	VT	T	F	VF	POST
VT													
PRE T					1				1				
F		4	1	1			4	1		4	5	2	
VF							1				1		

(ii) Listening to a typewriter for an hour or so would get/got on my nerves.

	VT	T	F	VF	VT	T	F	VF	VT	T	F	VF	POST
VT													
PRE T				1			2				2	1	
F			2	1			2	2			4	3	
VF			1	1				1			1	2	

(iii) Any method/this method of teaching is better than lecturing.

	VT	T	F	VF	VT	T	F	VF	VT	T	F	VF	POST
VT													
PRE T		2								2			
F	1	1	1		1	2	1		2	3	2		(1 abstension).
VF		1					2			1	2		

(iv) I learn/learned more from practical classes than from lectures.

	VT	T	F	VF	VT	T	F	VF	VT	T	F	VF	POST
VT													
PRE T		2	2			2	4			4	6		(1 abstension).
F		1	1							1	1		
VF													

(v) My typing is too slow for this method.

B

PRE \ POST	VT	T	F	VF
VT		1	1	
T		1	3	
F				
VF				

C

PRE \ POST	VT	T	F	VF
VT			1	
T		3	2	
F				1
VF				

total

PRE \ POST	VT	T	F	VF
VT		1	2	
T		4	5	
F				1
VF				

(vi) I would prefer a noiseless, swift, visual display to a typewriter, even though I would not get a hard copy.

PRE \ POST	VT	T	F	VF
VT				
T				
F			3	2
VF			1	

PRE \ POST	VT	T	F	VF
VT				1
T			1	
F			3	2
VF				

PRE \ POST	VT	T	F	VF
VT				1
T			1	
F			6	4
VF			1	

(vii) This method of teaching would be better if two people sat at one terminal.

PRE \ POST	VT	T	F	VF
VT				
T	1		2	1
F			1	
VF				1

PRE \ POST	VT	T	F	VF
VT				
T			3	
F		1	2	
VF				1

PRE \ POST	VT	T	F	VF
VT				
T	1		5	1
F		1	3	
VF				2

(viii) I (would) prefer to be taught in a definite sequence of topics rather than decide what I do next (group C only).

C

PRE \ POST	VT	T	F	VF
VT			1	
T		3	1	
F			1	1
VF				

APPENDIX D.

Part of a NUTS session which contains some source code in the author language from the course.

```
 *
cat
  PROGRAMS

  LESSONS
PILC0    RELEASED
PILC1    RELEASED
PILC2    RELEASED
PILC3    RELEASED
PILC4    RELEASED
PILC5    RELEASED
PILC6    RELEASED
PILC7    RELEASED

  RESPONSE FILES
PILC#0

END OF 'CAT'
 *
list,pilc3(2820,3590)


        CURRENT LISTING
 2820  Q13)TYPE('L OF (S3+S4+S5) = ?'5)
 2830  #CA0=<10>
 2840  N=0
 2850  RESP
 2860  N=N+1
 2870  IF(#CA0)JUMP>131
 2880  IF(#PERF(14NS))JUMP>132
 2890  IF(<NOT>#PERF(8NS))JUMP>133
 2900  TYPE('I ASSUME THEN IT IS NOT S4 BUT S5 YOU HAVEN''T GOT HOLD OF PROPERLY YET'1)
 2910  LOAD1>Q13
 2920  JUMP>Q8
 2930  133)TYPE('WRONG AGAIN. HERE IS THE ANSWER:'1)
 2940  TYPE('L OF (S3+S4+S5)=4+2+4=10.'1)
 2950  TYPE('NOW YOU SHOULD KNOW WHAT S3, S4 AND S5 ARE EXACTLY.'1)
 2960  JUMP>Q15
 2970  131)TYPE('EXCELLENT. YOU OBVIOUSLY KNOW ABOUT S3, S4 AND S5 EXACTLY.'1)
 2980  JUMP>Q15
 2990  132)TYPE('NO, BUT NEVER WORRY!! YOU CAN COME BACK TO THIS ONE. HOW ABOUT:'1)
 3000  Q14)TYPE('S4 = ?'5)
 3010  #CA0='"CC"'
 3020  #CA1='CC'
 3030  #CA2='""'K
 3040  N=0
 3050  RESP
 3060  N=N+1
 3070  IF(#CA0)JUMP>141
 3080  IF(#CA1<AND><NOT>#CA2)BACK('ENTER YOUR RESPONSE AGAIN, BUT USE DELIMITERS.'1)
```

```
3090  IF(N<EQ>1)BACK('YOU REMEMBER THAT S3=BCBC"?? LOOK AT DEFN. OF S4 THEN RETRY'1)
3100  IF(N<EQ>2)BACK('THE FIRST CHARACTER OF (3 $LC S3) =1 $FC "CBC"="C". HELP YOU?'1)
3110  TYPE('FROM WHAT YOU HAVE JUST BEEN TOLD, S4="C"+THE LAST CHARACTER OF S1'1)
3120  TYPE('="C"+"C"="CC".'1)
3130  142)TYPE('NOW GO BACK & TRY THAT QUESTION YOU WERE HAVING DIFFICULTY WITH.'1)
3140  JUMP>Q13
3150  141)TYPE('JOLLY GOOD.'1)
3160  JUMP>142
3170  Q15)TYPE('S1+"BC" $GE "A"+S5 = ?'5)
3180  #CA0='TRUE'
3190  #WA0='FALSE'
3195  #WA1='"'K
3200  RESP
3205  IF(#WA1)BACK('ANSWER AGAIN BUT WITHOUT STRING DELIMITERS!!'1)
3210  IF(#CA0)JUMP>151
3220  IF(<NOT>(#CA0<OR>#WA0))TYPE('COME ON!! THE ANSWER CAN ONLY BE ''TRUE'' OR'1)
3230  IF(<NOT>(#CA0<OR>#WA0))BACK(''''FALSE''.'1)
3240  TYPE('S1+"BC"="ABCBC" AND "A"+S5="ABCBC" SO THAT THE ANSWER IS TRUE.'1)
3250  JUMP>S5
3260  151)TYPE('GOOD, THAT''S YOUR LOT!!!!'1)
3270  S5)TYPE('NOW LET US LOOK AT 3 WORKED EXAMPLES.'5)
3280  PAUSE
3290  TYPE('FIRSTLY, SUPPOSE WE ARE GIVEN A STRING, S, WHOSE LENGTH IS AT LEAST'5)
3300  TYPE('4. WE ARE REQUIRED TO TAKE THE FIRST 4 CHARACTERS OF S AND PLACE THEM'1)
3310  TYPE('INTO A(1),A(2),A(3) & A(4),RESPECTIVELY, LEAVING THE REMAINING STRING'1)
3320  TYPE('IN S.'1)
3330  PAUSE
3340  TYPE('THERE ARE 4 BASIC STEPS TO THE SOLUTION:'5)
3350  TYPE('1. WE MUST OBTAIN A VALUE FOR S. THIS WE WILL ASSUME IS MERELY ''SET'''1)
3360  TYPE('TO A VALUE. (IF WE WERE GOING TO USE INDIRECT MODE, ''DEMAND'' WOULD'4)
3370  TYPE('MOST PROBABLY BE USED.)'4)
3380  TYPE('2. THE FIRST ELEMENT OF S IS PLACED INTO THE NEXT VACANT POSITION IN'1)
3390  TYPE('THE ARRAY.'4)
3400  TYPE('3. S IS REDUCED IN SIZE BY REMOVING THE FIRST CHARACTER.'1)
3410  TYPE('4. THE FINAL VALUES OF A AND S ARE PRINTED.'1)
3420  PAUSE
3430  TYPE('AS YOU WILL HAVE NO DOUBT REALISED, STEPS 2 & 3 ARE EACH OBEYED 4'5)
3440  TYPE('TIMES. THIS SUGGESTS A ''FOR'' STATEMENT. HOWEVER, AS BOTH STEPS HAVE'1)
3450  TYPE('TO BE OBEYED UNDER THE CONTROL OF THIS ''FOR'' STATEMENT, A SECOND'1)
3460  TYPE('''FOR'' MAY BE INTRODUCED TO PREVENT THE INTRODUCTION OF AN INDIRECT'1)
3470  TYPE('''PIL'' PART. REDUCING A STRING TO ''ITSELF MINUS THE FIRST CHARACTER'''1)
3480  TYPE('IS THE SAME AS ''THE LAST N CHARACTERS'' WHERE N IS THE CURRENT LENGTH'1)
3490  TYPE('MINUS ONE. SO WE HAVE:'1)
3500  PAUSE
3510  TYPE('=SET S="ABCDEFG".'3)
3520  TYPE('=FOR I=1 TO 4:FOR A(I)=1 $FC S:SET S=(L OF S -1) $LC S.'3)
```

```
 3530  TYPE('=TYPE A,S.'3)
 3540  TYPE('A(1) = "A"'4)
 3550  TYPE('A(2) = "B"'4)
 3560  TYPE('A(3) = "C"'4)
 3570  TYPE('A(4) = "D"'4)
 3580  TYPE('S = "EFG"'4)
 3590  TYPE(' '1)
END OF 'LIST'
 *
course,pilc
ENTER DIGIT TO DENOTE WHICH RESPONSE FILE TO USE
0

YOUR STARTING POSITION WILL BE LESSON 0 SEGMENT  4
IF YOU WISH TO CONTINUE FROM THERE JUST PRESS RETURN
OTHERWISE ENTER 'LESSON NO.,SEG. NO.' TO RESTART
3,1

ENTER '?END' IF YOU WISH TO FINISH NOW AT THIS PAUSE, OR OTHERWISE
HIT 'RETURN AS USUAL.
:

        CHARACTER STRINGS.
    ONE OF THE MORE POWERFUL FEATURES IN 'PIL' IS THE HANDLING OF CHARAC-
TER STRINGS. A CHARACTER STRING IS ANY SEQUENCE OF CHARACTERS (INCLUD-
ING A NULL SEQUENCE). A CHARACTER STRING CONSTANT IS ANY STRING ENCLOS-
ED IN DOUBLE QUOTATION MARKS (").
  =SET A="A STRING OF CHARACTERS".
  =SET NULL="".
:

    ANY LEGITIMATE INPUT CHARACTERS MAY BE INCLUDED IN A STRING, EXCEPT
FOR ("), WHICH MAY ACT ONLY AS A DELIMITER IN A CHARACTER STRING
CONSTANT AND IS NOT PART OF THE STRING. AN UPPER LIMIT OF 255 CHARACT-
ERS IN A SINGLE STRING IS IMPOSED.
:

    STRING COMPARISON.

    ANY STRING MAY BE COMPARED WITH ANY OTHER STRING, USING ANY OF THE
DEFINED RELATIONAL OPERATORS. STRINGS ARE COMPARED LEFT TO RIGHT. IF
STRINGS ARE OF AN UNEQUAL LENGTH, THE SHORTER STRING IS TREATED AS THO
IT WERE PADDED AT THE RIGHT END WITH BLANKS FOR COMPARISON.
:

    THE FOLLOWING COLLATING SEQUENCE IS THE BASIS FOR COMPARISON OF
STRINGS:
    (BLANK)<PUNCTUATION<a,b,..,z<A,B,..Z<0,1,..,9
E.G.
  =IF "X<"Y",TYPE "YES".
   YES
  =IF "ABCD"="ABCD  ",TYPE "BLANKS IGNORED".
   BLANKS IGNORED
:
?end
*
```

APPENDIX E.

Part of a student session from the course.

```
ENTER '?END' IF YOU WISH TO FINISH NOW AT THIS PAUSE, OR OTHERWISE
HIT 'RETURN AS USUAL.
:

        CHARACTER STRINGS.
    ONE OF THE MORE POWERFUL FEATURES IN 'PIL' IS THE HANDLING OF CHARAC-
TER STRINGS. A CHARACTER STRING IS ANY SEQUENCE OF CHARACTERS (INCLUD-
ING A NULL SEQUENCE). A CHARACTER STRING CONSTANT IS ANY STRING ENCLOS-
ED IN DOUBLE QUOTATION MARKS (").
  =SET A="A STRING OF CHARACTERS".
  =SET NULL="".
:

    ANY LEGITIMATE INPUT CHARACTERS MAY BE INCLUDED IN A STRING, EXCEPT
FOR ("), WHICH MAY ACT ONLY AS A DELIMITER IN A CHARACTER STRING
CONSTANT AND IS NOT PART OF THE STRING. AN UPPER LIMIT OF 255 CHARACT-
ERS IN A SINGLE STRING IS IMPOSED.
:

    STRING COMPARISON.

    ANY STRING MAY BE COMPARED WITH ANY OTHER STRING, USING ANY OF THE
DEFINED RELATIONAL OPERATORS. STRINGS ARE COMPARED LEFT TO RIGHT. IF
STRINGS ARE OF AN UNEQUAL LENGTH, THE SHORTER STRING IS TREATED AS THO
IT WERE PADDED AT THE RIGHT END WITH BLANKS FOR COMPARISON.
:

    THE FOLLOWING COLLATING SEQUENCE IS THE BASIS FOR COMPARISON OF
STRINGS:
    (BLANK)<PUNCTUATION<a,b,..,z<A,B,..Z<0,1,..,9
E.G.
  =IF "X<"Y",TYPE "YES".
   YES
  =IF "ABCD"="ABCD  ",TYPE "BLANKS IGNORED".
   BLANKS IGNORED
:

    HOWEVER, IF YOU REMEMBER, 'a,b,..,z' ARE NOT AVAILABLE TO YOU IN
THIS COURSE. IF YOU TYPE THEM, THEY ARE AUTOMATICALLY CONVERTED TO
'A,B,C,...,Z' RESPECTIVELY.
:

    IT IS ASSUMED THAT BLANK IS THE LOWEST CHARACTER THAT A STRING WILL
CONTAIN.
    IN THE FOLLOWING BOOLEAN EXPRESSIONS WHICH INVOLVE STRING COMPARISON
, ENTER 'T' OR 'F' DEPENDING ON WHETHER YOU THINK THE VALUE OF THE
EXPRESSION IS TRUE OR FALSE.
    "GREATER"<"GREATEST"
t
GOOD, AN EASY ONE TO BEGIN WITH.
    " APL975" $GT "APL97$" $AND "A21X" = "A21X  "
```

```
no idea!!
ENTER 'T' OR 'F'.
t
THIS IS MORE TRICKY. IN AN ATTEMPT TO FOOL YOU, THE FIRST STRING BEGINS
WITH A BLANK WHILE THE SECOND ONE DOES NOT. HENCE THE FIRST CONDITION
IS FALSE. THUS, AS THE RESULTS ARE TO BE 'ANDED', IT DOES NOT MATTER
WHAT THE SECOND CONDITION TURNS OUT TO BE. AS IT HAPPENS, TRAILING
BLANKS ARE IGNORED SO IT IS TRUE, BUT THE WHOLE EXPRESSION IS FALSE.
    "1000">"999"
f
YES, THE LEADING CHARACTER TELLS ALL!!
    "YORKS." $GE "YORKSHIRE"
f
PERFECT-A GOOD WAY TO FINISH OFF A SEQUENCE OF QUESTIONS.
         STRING OPERATIONS & FUNCTIONS

    HERE IS A TABLE OF STRING OPERATORS AND FUNCTIONS AVAILABLE TO YOU
IN 'PIL'. AS WITH THE ARITHMETRIC FUNCTIONS THERE ARE LONG AND SHORT
FORMS TO SOME OF THESE.
:

         OPERATOR                              MEANING
 SHORT                LONG
   +                                    CONCATERNATION
   "                                    STRING DELIMITER
  L OF     THE LENGTH OF                LENGTH OF A CHARACTER STRING
N $LC A    THE LAST N CHARACTERS OF A   DECONCATERNATION
N $FC A    THE FIRST N CHARACTERS OF A  DECONCATERNATION
           THE VALUE OF                 EVALUATE CONTENTS AS A 'PIL' EXPRESSION
           THE BCD VALUE OF             CONVERTS RADIX OF ALL OPERANDS TO STRING
:

    TWO STRINGS MAY BE CONCATERNATED, I.E. THE SECOND JOINED TO THE END
OF THE FIRST. THE '+' (PLUS) OPERATOR PERFORMS THIS TASK, PROVIDED
THAT BOTH OPERANDS ARE OF STRING MODE. THE LENGTH OF THE CONCATERNATION
RESULT IS THE SUM OF THE LENGTHS OF THE 2 OPERANDS. TO ILLUSTRATE:
  =SET X="1234"
  =SET Y="567890"
  =SET Z=X+Y+"ABC"
  =TYPE Z,THE LENGTH OF Z.
   Z="1234567890ABC"
   THE LENGTH OF Z = 13.0
:

    YOU WILL HAVE NOTICED THE INTRODUCTION OF THE LENGTH FUNCTION.
TO DETERMINE THE LENGTH OF A STRING, 'THE LENGTH OF' (OR 'L OF')
FUNCTION IS USED. ITS VALUE, A NUMERIC, IS THE COUNT OF THE
CHARACTERS CONTAINED IN THE GIVEN STRING. IT WILL ALWAYS BE AN INTEGER
IN THE RANGE 0 TO 255.
  =TYPE THE LENGTH OF "",L OF ("AB"+"BC").
   THE LENGTH OF "" = 0.0
   L OF ("AB"+"BC") = 4.0
    STRING SUBTRACTION IS NOT WELL DEFINED, AND IS THEREFORE NOT
ALLOWED. IT IS USEFUL, HOWEVER, EITHER TO REMOVE OR EXAMINE SOME
PORTION OF A LONG STRING.
:
```

```
    TO OBTAIN SUCH SUBSTRINGS FROM STRINGS, 'PIL' PROVIDES 2 FUNCTIONS
NAMED:
  THE FIRST N CHARACTERS OF S
  THE LAST N CHARACTERS OF S
WHERE N IS ANY ARITHMETIC EXPRESSION AND S IS A STRING.
:

    THE ABBREVIATIONS FOR THESE FUNCTIONS ARE, RESPECTIVELY,
  N $FC S
  N $LC S
TO GET THE FIRST OR LAST CHARACTER OF A STRING, S, ONE MAY WRITE 1 FOR
N IN THE ABOVE OR USE THE NAMES:
  THE FIRST CHARACTER OF S
  THE LAST CHARACTER OF S
:

    EACH FUNCTION IS SELF-EXPLANATORY. THE NUMBER SPECIFIED MUST BE
NON-NEGATIVE, AND NOT GREATER THAN THE LENGTH OF THE STRING TO BE
OPERATED ON.
E.G. 3 $FC "ABCDEFG" = "ABC"
     2 $LC "12345" = "45"
:

    COMBINATIONS OF 'THE FIRST' AND 'THE LAST' ALLOW EXAMINATION AT ANY
POINT WITHIN A STRING:
  =SET X=THE FIRST CHARACTER OF THE LAST 3 CHARACTERS OF "ABCDEFG".
  =TYPE X.
   X = "E"

GIVEN THE FOLLOWING SEQUENCE OF 'PIL' INSTRUCTIONS:
  =SET S1="ABC".
  =SET S2="BCDEF".
  =SET S3=2 $LC S1 + (L OF S1 -1) $FC S2.
  =SET S4=1 $FC 3 $LC S3 + THE LAST CHARACTER OF S1.
  =SET S5=THE FIRST 1 CHARACTERS OF S2 + 1 $FC 4 $LC S2 +
                        ((L OF S2)-(L OF S1)) $FC S3.
  =SET M=L OF S2 + L OF S5.
  =SET N=L OF ((L OF S3) $FC S2).
  =SET B1=S3=S5.
  =SET B2=S3+S4<S2+"BC".
ENTER THE VALUE OF THE FOLLOWING EXPRESSIONS (DON'T FORGET TO ENTER
THE DELIMITERS OF A STRING WHEN THE ANSWER IS A STRING!!!!
YOU NEEDN'T THINK TOO LONG OVER THESE QUESTIONS-YOU WILL
GET HELP IF YOU SIMPLY HIT 'RETURN'.
    S3 = ?
abbc
(L OF S1-1) IS SIMPLY (3-1)=2. NOW TRY AGAIN.
bcbc
TRY RE-ENTERING YOUR ANSWER WITH STRING DELIM.S
"bcbc"
CORRECT. S3="BCBC".
    B1 = ?
help??
B1 IS A LOGICAL,YOU KNOW!!!!
NEVER MIND ABOUT THAT QUESTION-WE'LL COME BACK TO IT. TRY THIS
ONE FIRST.
    S5 = ?
"bcbb"
```

```
NO. L OF S2=5 & L OF S1=3 SO THAT (5-3) $FC S3="BC". NOW TRY.
"bccb"
NO,AGAIN. 1 $FC 4 $LC S2 =1 $FC "CDEF"="C".TRY AGAIN, S5 = ?
"bccc"
STILL NO!! THE FIRST 1 CHARACTERS OF S2 ="B". NOW, SURELY!!
"bcbc"
THAT'S THE ONE!!
NOW, WHAT ABOUT THE QUESTION YOU WERE HAVING TROUBLE WITH!!
    B1 = ?
true
GOOD, YOU OBVIOUSLY KNOW THAT S5="BCBC", TOO.
    THE SQUARE ROOT OF (M*M+9*N**2) = ?
don't know
NO,SORRY, O.K., TRY THIS EASIER QUESTION FIRST BEFORE YOU TRY AGAIN.
    N = ?
4
QUITE CORRECT. NOW BACK TO THE SQUARE ROOT QUESTION.
    THE SQUARE ROOT OF (M*M+9*N**2) = ?
5
YOU NOW KNOW WHAT N IS 4 & M=5+4=9. SO, ANSWER = ?
15
GOOD, YOU OBVIOUSLY KNOW THAT M=9 AND N=4.
    B1 & $NOT B2 = ?
true
GOOD. YOU'RE DEAD RIGHT THAT B2 IS FALSE.
    L OF (S3+S4+S5) = ?
10
EXCELLENT. YOU OBVIOUSLY KNOW ABOUT S3, S4 AND S5 EXACTLY.
    S1+"BC" $GE "A"+S5 = ?
false
S1+"BC"="ABCBC" AND "A"+S5="ABCBC" SO THAT THE ANSWER IS TRUE.
    NOW LET US LOOK AT 3 WORKED EXAMPLES.
:

    FIRSTLY, SUPPOSE WE ARE GIVEN A STRING, S, WHOSE LENGTH IS AT LEAST
4. WE ARE REQUIRED TO TAKE THE FIRST 4 CHARACTERS OF S AND PLACE THEM
INTO A(1),A(2),A(3) & A(4),RESPECTIVELY, LEAVING THE REMAINING STRING
IN S.
:

    THERE ARE 4 BASIC STEPS TO THE SOLUTION:
1. WE MUST OBTAIN A VALUE FOR S. THIS WE WILL ASSUME IS MERELY 'SET'
   TO A VALUE. (IF WE WERE GOING TO USE INDIRECT MODE, 'DEMAND' WOULD
   MOST PROBABLY BE USED.)
2. THE FIRST ELEMENT OF S IS PLACED INTO THE NEXT VACANT POSITION IN
   THE ARRAY.
3. S IS REDUCED IN SIZE BY REMOVING THE FIRST CHARACTER.
4. THE FINAL VALUES OF A AND S ARE PRINTED.
:
```

```
    AS YOU WILL HAVE NO DOUBT REALISED, STEPS 2 & 3 ARE EACH OBEYED 4
TIMES. THIS SUGGESTS A 'FOR' STATEMENT. HOWEVER, AS BOTH STEPS HAVE
TO BE OBEYED UNDER THE CONTROL OF THIS 'FOR' STATEMENT, A SECOND
'FOR' MAY BE INTRODUCED TO PREVENT THE INTRODUCTION OF AN INDIRECT
'PIL' PART. REDUCING A STRING TO 'ITSELF MINUS THE FIRST CHARACTER'
IS THE SAME AS 'THE LAST N CHARACTERS' WHERE N IS THE CURRENT LENGTH
MINUS ONE. SO WE HAVE:
:

  =SET S="ABCDEFG".
  =FOR I=1 TO 4:FOR A(I)=1 $FC S:SET S=(L OF S -1) $LC S.
  =TYPE A,S.
   A(1) = "A"
   A(2) = "B"
   A(3) = "C"
   A(4) = "D"
   S = "EFG"

    NOW FOR OUR SECOND WORKED EXAMPLE.
HERE IT IS:
    READ IN A CHARACTER STRING AND PRINT IT IN REVERSE ORDER.
:

    HERE AGAIN THE PROBLEM CAN BE BROKEN DOWN INTO 4 STAGES.
1. WE MUST READ IN THE STRING,S,SAY.
2. THE REVERSE STRING, R, SAY, IS SET TO THE NULL STRING.
3. WORKING FROM BACK TO FRONT THROUGH S, ONE CHARACTER AT A TIME IS
   ADDED TO R.
4. R IS PRINTED OUT.
:

STAGE 1 IS EASILY PERFORMED USING A 'DEMAND' INSTRUCTION. THE NULL
STRING IS SIMPLY 2 CONSECUTIVE DELIMITERS. STAGE 3 MUST BE CARRIED OUT
THE SAME NUMBER OF TIMES AS THE LENGTH OF S. THIS SUGGESTS A 'FOR'
STATEMENT 'TO L OF S'. TO PICK UP THE CHARACTER TO ADD TO R, 'THE
FIRST CHARACTER OF THE LAST I CHARACTERS OF S' IS USED, ASSUMING THAT
I IS THE 'FOR' VARIABLE.
:

  =1.1 DEMAND S.
  =1.2 SET R="".
  =1.3 FOR I=1 TO L OF S:SET R=R+1 $FC I $LC S.
  =1.4 TYPE R.
  =DO PART 1.
   S =
   "QWERT"
   R = "TREWQ"
EASY, ISN'T IT!!!!

    AND SO TO OUR LAST WORKED EXAMPLE IN THESE 3.
BUT FIRST LET ME EXPLAIN ONE SMALL 'PIL' FUNCTION YOU NEED TO KNOW
FOR THIS EXAMPLE.
:
```

```
    IT IS THE 'SWAP' STATEMENT, WHICH INTERCHANGES THE VALUES AND MODES
OF 2 VARIABLES.
  =SET A=3.
  =SET B=1<2.
  =SWAP A,B.
  =TYPE A,B.
   A = TRUE
   B = 3
:

    THIS AFFECTS A & B IN THE SAME WAY (BUT MORE EFFICIENTLY) AS THE
SEQUENCE:
  =SET TEMP=A.
  =SET A=B.
  =SET B=TEMP.
:

    NOW TO THE EXAMPLE:
IF THE VECTOR WORDS(I) FOR I=1,2,...N, CONTAINS A SET OF N STRINGS,
WRITE A PROGRAM TO PLACE THESE STRINGS INTO ALPHABETIC ORDER IN WORDS.
:

    THE METHOD TO USE FOR THIS ONE MAY BE 'THE OLD WIFE'S SORT'.
THE FIRST ALPHABETICAL WORD IS FOUND IN THE ARRAY THEN SWAPPED WITH THE
FIRST ELEMENT. THEN THE NEXT ALPHABETICAL WORD IS FOUND AND SWAPPED
WITH THE SECOND WORD. AND SO ON.
:

  =8.1 FOR I=1 TO N-1:DO PART 9.
  =9.1 SET J=I.
  =9.2 FOR K=I+1 TO N:IF WORDS (J)>WORDS(K),SET J=K.
  =9.3 SWAP WORDS(I),WORDS(J).
WHICH, WITH A TEST RUN, MAY GIVE:
  =SET WORDS(1)="DOG".
  =SET WORDS(2)="CAT".
  =SET WORDS(3)="360/67".
  =SET WORDS(4)="CAMEL".
  =DO PART 8.
  =TYPE WORDS.
   WORDS(1) = "CAMEL"
   WORDS(2) = "CAT"
   WORDS(3) = "DOG"
   WORDS(4) = "360/67"

AND THERE WE ARE!!

    THERE ARE SOME ADDITIONAL STRING FUNCTIONS TO 'PIL'.
THE MOST UNUSUAL IS 'THE VALUE' FUNCTION.
:

    IT IS DEFINED AS FOLLOWS: IF THE MODE OF THE OPERAND IS STRING, THIS
STRING IS EVALUATED AS A 'PIL' EXPRESSION. IF THE MODE IS NOT STRING,
THE RESULT IS THE SAME AS THE OPERAND. ONE USE FOR THIS FUNCTION IS
CONVERTING A STRING CONTAINING NUMERIC DIGITS TO INTERNAL NOTATION.
:
```

```
E.G.
  =SET A=3.
  =SET B=5.
  =SET C="A+B**2".
  =TYPE THE VALUE OF C.
   THE VALUE OF C = 13.0
  =TYPE THE VALUE OF "12345".
   THE VALUE OF "12345" = 12345.0
:

    'THE BCD VALUE' FUNCTION ALLOWS CONVERSION IN THE OTHER DIRECTION.
IF ANY OPERAND IS NUMERIC, THE RESULT WILL BE A STRING OF DIGITS
IDENTICAL TO THE WAY THE NUMBER WOULD LOOK IF TYPED OUT WITH A LENGTH
OF 14. IF THE OPERAND IS STRING. THE 'BCD VALUE' IS IDENTICAL TO THE
OPERAND. IF THE OPERAND IS BOOLEAN, THE 'BCD VALUE' WILL BE EITHER
"TRUE" OR "FALSE".
:

E.G.
  =SET A=3.
  =TYPE THE BCD VALUE OF (A*A),THE BCD VALUE OF (A>4).
   THE BCD VALUE OF (A*A) = " 9.0          "
   THE BCD VALUE OF (A>4) = "FALSE"
    THE FORMAT OF THE RESULT OF THE VALUE IS ALWAYS THE SAME AS THAT
GENERATED BY 'TYPE'.
:

    GIVEN THE INSTRUCTIONS:
  =SET A=10.
  =SET B=0.
  =SET C=4.
  =SET D="A*SQRT OF C+5".
  =SET E="B $LE 0 $AND A>C".
ENTER THE VALUES OF THE FOLLOWING:
    THE VALUE OF C = ?
" 4.0          "
COME ALONG!! IT'S MUCH EASIER THAN IT LOOKS. TRY AGAIN.
4
GOOD. FOR A NUMERIC, THE VALUE IS ITSELF.
    THE VALUE OF D = ?
25
QUITE CORRECT. IT'S EASY, REALLY, SO LONG AS YOU KNOW HOW TO
EVALUATE EXPRESSIONS.
    THE VALUE OF E = ?
true
JOLLY GOOD. THE ANSWER IS A STRAIGHTFORWARD BOOLEAN.
    THE BCD VALUE OF (A*COS OF B) = ?
10
FOR A START, THE ANSWER SHOULD BE A STRING!! TRY AGAIN.
"10.0          "
A*COS OF B =10*1=10. THE BCD VALUE IS THUS " 10.0          ".
    HERE IS AN EXAMPLE ON THE USE OF 'THE VALUE OF '.
:

READ IN A CHARACTER STRING, S, WHERE THE FIRST TWO CHARACTERS REPRESENT
A DECIMAL INTEGER, I,SAY, WHILE THE THIRD AND FOURTH CHARACTERS ALSO
REPRESENT A DECIMAL INTEGER, J,SAY. PRINT OUT THE SUBSTRING, R, OF
LENGTH J BEGINNING AT THE I TH POSITION OF THE STRING READ IN.
E.G. "0804ABCDEFGHIJK" GIVES "DEFG".
:
```

```
    THE POINTS TO REMEMBER HERE ARE:
1. I WILL BE THE VALUE OF THE FIRST 2 CHARACTERS OF S.
2. J WILL BE THE VALUE OF THE LAST  2 CHARACTERS OF THE FIRST 4
   CHARACTERS OF S.
3. THE SUBSTRING WILL BE THE LAST J CHARACTERS OF THE FIRST (I+J-1)
   CHARACTERS OF S.
:

  =1.1 DEMAND S.
  =1.2 SET I=THE VALUE OF 2 $FC S.
  =1.3 SET J=THE VALUE OF 2 $LC 4 $FC S.
  =1.4 SET R=J $LC (I+J-1) $FC S.
  =1.5 TYPE R.
  =DO PART 1.
   S =
   "0804ABCDEFGHIJK"
   R = "DEFG"

QUITE STRAIGHTFORWARD, DON'T YOU THINK!!!!
    ENOUGH OF STRINGS. NOW LET US LOOK AT SOME MORE INPUT/OUTPUT.
:

        EXTENDED CONSOLE I/O
    THERE IS A METHOD BY WHICH THE USER MAY CONTROL THE FORMAT OF AN OUTPUT
OR INPUT LINE, ALLOWING SPECIFICATION OF ANY NUMBER OF ITEMS ON A
SINGLE LINE.
:

    THE PERTINENT STATEMENTS ARE:
         TYPE IN FORM N,LIST.
         DEMAND IN FORM N,LIST.
WHERE LIST IS THE SAME AS FOR THE NORMAL 'TYPE' AND 'DEMAND'
STATEMENTS, AND N IS ONE OF THREE ELEMENTS AS FOLLOWS:
:

 1. A STRING CONSTANT (LITERAL) TO BE USED AS A 'FORM'.
 2. AN ARITHMETIC EXPRESSION USED TO REFERENCE A 'FORM' ALREADY
    DEFINED.
 3. A VARIABLE WHOSE CONTENT IS A STRING TO BE USED AS A 'FORM'.
:

    THUS, THE FOLLOWING ARE EQUIVALENT:
  =TYPE IN FORM "####",LIST.  (1,ABOVE)
    OR
  =FORM 1.
   ####
  =TYPE IN FORM 1,LIST.       (2,ABOVE)
    OR
  =SET X="####".
  =TYPE IN FORM X,LIST.       (3,ABOVE)
:
```

AS CAN BE SEEN, A 'FORM' MAY BE DEFINED AS A NORMAL STRING, EITHER IN A VARIABLE OR IN THE 'TYPE' OR 'DEMAND' STATEMENT, OR BY THE 'FORM' STATEMENT. THE FORM STATEMENT HAS THE FORMAT:
=FORM N.
WHERE N IS AN INTEGER FROM 1 TO 4 DIGITS LONG.
:

THE NEXT INPUT LINE AFTER THE 'FORM'
STATEMENT IS TAKEN AS THE FORM ITSELF; FORMS MAY BE DEFINED IN DIRECT MODE ONLY, SINCE THE 'FORM'
STATEMENT MAY OCCUR IN DIRECT MODE ONLY. NOTE THAT A TERMINAL PERIOD IS NOT REQUIRED FOR A FORM.
:

STUDY THE FOLLOWING STATEMENTS:
```
=SET I=2.
=SET J=3.
=SET K=1.
=FORM 4.
=##
=FORM 7.
=###
=SET X="###".
=SET Y="####".
=SET Z="##".
```
:

ARE THE FOLLOWING 2 STATEMENTS EQUIVALENT? ENTER 'YES' OR 'NO'.
```
=TYPE IN FORM I*J+K-J*(I-K-1),LIST.
=TYPE IN FORM Z+"#",LIST.
```
yes
GOOD. WOULD YOU LIKE TO SAY WHAT THE FORM IS?
###
YES, '###',FORM 7, IS CORRECT.
ARE THESE 2 EQUIVALENT? AGAIN ENTER 'YES' OR 'NO'.
```
=TYPE IN FORM X,LIST.
=TYPE IN FORM (J $FC (L OF Z+1) $LC Y),LIST.
```
no
NO, THEY'RE THE SAME ONCE AGAIN.
THE FORMER IS FORM X, WHICH IS '###'.
AND THE LATTER IS FORM (3 $FC (2+1) $LC "####" = 3 $FC 3 $LC "####"
="###". YOU SHOULD BE WITH THIS NOW!!
OK. ONE RIGHT, ONE WRONG. LET'S FINISH OFF WITH A WINNER!!
HOW ABOUT THIS PAIR?
```
=TYPE IN FORM (L OF X +K),LIST.
=TYPE IN FORM THE VALUE OF "L OF X + L OF Y",LIST.
```
ARE THEY EQUIVALENT? ENTER 'YES' OR 'NO'.
no
SPOT ON. THE FIRST ONE GIVES THE FORM '##' BUT THE SECOND ONE GIVES '###'.
BEFORE WE CONTINUE TO DISCUSS THE DIFFERENT TYPES OF FORM, LET ME REMIND YOU TO TAKE PARTICULAR CARE WITH THE ENTRY OF A MEANINGFUL '_' AND A MEANINGFUL '!' TO THIS TERMINAL.
IT IS MENTIONED IN YOUR NOTES.

A FORM SPECIFIES THE FORMAT OF THE LINE TO BE READ OR PRINTED. THE FORM ITSELF IS A STRING OF CHARACTERS. HERE ARE THE MEANINGS OF SOME OF THESE.
:

NUMERIC
1. STANDARD NUMERIC FIELD.

```
    A STANDARD NUMERIC FIELD IS REPRESENTED BY A SERIES OF UNDERLINE
CHARACTERS AND AN OPTIONAL DECIMAL POINT.
:

    EACH UNDERLINE INDICATES A POSSIBLE DIGIT POSITION, LIMITED BY THE
NUMBER OF ALLOWABLE SIGNIFICANT DIGITS IN A 'PIL' NUMBER . AT LEAST
ONE HIGH ORDER POSITION SHOULD BE SPECIFIED IN ORDER TO ACCOMODATE A
POSSIBLE MINUS SIGN.
    E.G. ( THE 'TYPE FORM' STATEMENT USED HERE CAN BE USED EITHER TO
          EXAMINE A FORM FOR ERRORS, OR TO TYPE OUT A HEADER LINE
          ENTERED AS A FORM. NO IDENTIFICATION IS TYPED WITH THE FORM.)
  =FORM 3.
  =!_!_.!_!_!_
  =TYPE FORM 3.

  __.___
  =TYPE IN FORM 3,1.2376.
    1.237
:  .

    NOTICE THAT ADDITIONAL HIGH ORDER POSITIONS ARE LEFT BLANK, WHILE THE
NUMBER IS TRUNCATED(NOT ROUNDED)AFTER THE NUMBER OF ALLOWABLE
DIGITS TO THE RIGHT OF THE DECIMAL POINT.
:

    HOW WOULD THE NUMBER '.06789' APPEAR IF TYPED IN THE ABOVE FORM?
  .067
VERY NEARLY. WHY A SPACE BEFORE THE DECIMAL POINT?? TRY AGAIN.
 0.067
EXCELLENT. EXACTLY RIGHT.
    NOW TRY THE NUMMER '-1.1' IN THE SAME FORM.
-1.100
PERFECT. RIGHT ALL THE WAY!!
BOTH RIGHT-YOU UNDERSTAND THAT FORM OK.
    2. SCIENTIFIC NOTATION.
    THIS MAY BE REQUESTED IN A SPECIAL 'FORM' SPECIFICATION.
E.G.
  =FORM 2.
  =.........
  =TYPE FORM 2.
   .........
  =TYPE IN FORM 2,-.123456.
   -1.234E-01
:

AS YOU WILL NOTICE, SCIENTIFIC NOTATION GIVES THE GREATEST NUMBER OF
SIGNIFICANT FIGURES POSSIBLE FOR THE NUMBER OF PERIODS WE ALLOTTED.
:

    AT LEAST 7 CONSECUTIVE PERIODS ARE REQUIRED FOR 1 SIGNIFICANT DIGIT
OF OUTPUT (USING SCIENTIFIC NOTATION), EACH ADDITIONAL PERIOD ALLOWING
ONE MORE DIGIT OF SIGNIFICANCE, UP TO THE LIMITATION OF 'PIL' NUMBERS
ANY NUMBER CAN BE TYPED OUT SUCCESSFULLY IN THIS FORM.
:
```

```
+3.141593e+00
    GIVEN THE FORM '.....' HOW WOULD THE NUMBER '3.141593'
APPEAR IF TYPED IN THIS FORM?
+3.141593e+00
WHEN POSITIVE, A BLANK APPEARS INSTEAD OF '+'.
12 PERIODS,LESS 7 FOR SIGNS,EXPONENT ETC. LEAVES 5,NOT 6, FOR
THE RHS!!
TRY AGAIN.
 3.14159e+00
EXCELLENT. YOU'VE GOT IT EXACTLY RIGHT.
    WHAT ABOUT THE NUMBER '-12345.67' IN THIS SAME FORM?
1.23457e+04
YOU'VE CORRECTLY GIVEN ONLY 5 PLACES,BUT ROUNDED,NOT TRUNCATED
PLEASE TRY AGAIN.
1.23456e+04
NO. THE ANSWER IS: '-1.23456E+04'.
HAVING GOT THE LAST ONE RIGHT, YOU SHOULD HAVE HAD THIS ONE, TOO!!
THE 'FLOATING POINT' IDEA ISN'T TOO DIFFICULT, REALLY.

    NUMBERS WHICH ARE TOO LARGE FOR STANDARD NUMERIC FORMS WILL GENERATE
A DIAGNOSTIC MESSAGE. SUCH NUMBERS MAY BE TYPED IN SCIENTIFIC NOTATION
OR TYPED IN THE FOLLOWING SPECIAL NOTATION.
  =FORM 6.
  =!_!_.!_!_!_!!!!!!!!
  =TYPE FORM 6.
   __.___!!!!
:

    THE FORM IS A STANDARD NUMERIC FORM FOLLOWED BY 4 (BUT TO YOU, REALLY
8,REMEMBER?) EXCLAMATION MARKS. THE FORM WILL TYPE OUT STANDARD NUMERIC
WHEN THE NUMBER IS WITHIN RANGE OF THE SPECIFICATION, BUT WILL SWITCH
TO SCALED NOTATION WHEN THE NUMBER IS TOO LARGE OR TOO SMALL. AS WITH
SCIENTIFIC NOTATION, ALL NUMBERS CAN BE TYPED OUT IN THIS FORM.
:

    CONSIDER THE FOLLOWING FORM:
  =FORM 23.
  =!_!_.!_!_!_!!!!!!!!!_!_!_.!_!_!!!!!!!!
  =TYPE FORM 23.
   __.___!!!!___.__!!!!
  =TYPE IN FORM 23,3.14159,1327.6.
    3.141    132.76E+01

YOU WILL NOTICE THAT 4 SPACES SEPERATE THE 2 NUMBERS AS AN EXPONENT WAS
NOT REQUIRED FOR THE FIRST ONE.
        ALPHABETIC INFORMATION.

    THE CHARACTER '#' INDICATES ONE POSITION OF ALPHABETIC INFORMATION.
OPERANDS IN DOUBLE QUOTES, STRING VARIABLES, OR BOOLEAN EXPRESSIONS
WILL TYPE IN THIS FORM.
:
```

```
    E.G.
  =FORM 10.
  =## !_!_ ####
  =TYPE FORM 10.
   ## __ ####
  =TYPE IN FORM 10,"X",10,"STRING".
   X  10 STRI
:
```

AS CAN BE SEEN IN THE ABOVE EXAMPLE, THERE IS A SPACE AFTER THE FIRST ALPHABETIC FIELD AND A SPACE AFTER THE NUMERIC FIELD. IN FACT, ANY CHARACTERS ARE ALLOWED IN A FORM. THEY WILL BE OUTPUT WHEN THE FORM IS USED IN A 'TYPE' STATEMENT.

:

```
    E.G.
  =FORM 9.
  =X=!_!_!_ Y=!_!_!_.!_!_
  =TYPE FORM 9.
   X=___ Y=___.__
  =TYPE IN FORM 9,X,Y.
   X=537 Y= 49.83
```

NOW LET US RETURN TO THE 'TYPE IN FORM N,LIST.' STATEMENT. SHOULD THE LIST CONTAIN MORE ITEMS THAN THE FORM ALLOWS, THE FORM WILL BE RESCANNED FROM THE BEGINNING UNTIL ALL ITEMS IN THE LIST HAVE BEEN TYPED. E.G.

```
  =FORM 8.
  =!_!_ !_!_
  =TYPE FORM 8.

  =TYPE IN FORM 8,I,J,K.       (WHERE I=9,J=10,K=11)
    9 10
    11
:
```

CONSIDER THE FOLLOWING SEQUENCE OF 'PIL' STATEMENTS:

```
  =FORM 326.
  =!_!_ !_!_ !_!_
  =TYPE FORM 326.

  =FOR I=1 TO 3: FOR J=1 TO 3:SET A(I,J)=I*10+J.
  =TYPE IN FORM 326,A.
```

WHAT WILL THE NEXT LINE (AN OUTPUT LINE) OF THIS SEQUENCE BE?

```
11 12 13
GOOD. THE REST SHOULD BE A CAKE-WALK!
    WELL,THEN,GIVE ME THE NEXT OUTPUT LINE.
21 22 23
```

```
YES-YOU'VE GOT IT RIGHT!!
    AND FINALLY, WHAT IS THE LAST LINE RESULTING FROM THE 'TYPE'
STATEMENT?
31 32 33
CORRECT. JUST THE WAY TO FINISH A SEQUENCE.

    WE HAVE PREVIOUSLY MENTIONED THE STATEMENT 'DEMAND IN FORM N,LIST.'
HOWEVER, THE FOLLOWING NOTES AND RESTRICTIONS APPLY:
:

1. NUMERIC FORM FIELDS MERELY INDICATE THAT NUMERIC INPUT IS EXPECTED
   AND NO ALIGNMENT OF INPUT TO DECIMAL POINTS IS NECESSARY, AND NO
   SCALING IS PERFORMED.
:

2. THE CHARACTER " (DOUBLE QUOTE) IS NOT AN ACCEPTABLE INPUT CHARACTER
   IN AN ALPHABETIC FORM FIELD.
:

3. THE FORM FIELDS DRIVING THE INPUT LINE MUST BE COMPLETELY SATISFIED,
   IT IS AN ERROR IF A 'DEMAND IN FORM' STATEMENT WITH 4 INPUT PARAMS.
   ,COUPLED WITH A FORM WITH 4 FIELDS, RECEIVES ONLY 3 INPUT ITEMS ON A
   LINE.
:

    AND FINALLY.....
EXTENDED I/O LISTED FEATURES.
THERE IS A WAY BY WHICH A 'FOR' CAN OPERATE WITHIN BOTH 'DEMAND'
AND 'TYPE' STATEMENTS IN THE STANDARD I/O LISTS.
E.G.
  =1.8 TYPE (FOR I=1 TO 5: A(I),B(I)).
  =1.9 DEMAND (FOR I=1 TO 5: A(I)).
:

    THIS EXTENSION IS MOST USEFUL IN CONJUNCTION WITH USER DIRECTED INPUT
AND OUTPUT, AS IT ALLOWS SPECIFICATION OF SEVERAL ITEMS IN AN ARRAY
WITHOUT LISTING THEM INDIVIDUALLY. THE STANDARD RULES FOR 'FOR' APPLY
,INCLUDING NESTING.
:

    CONSIDER THIS EXAMPLE:
  =FOR I=1 TO 5:FOR J=1 TO 5:SET A(I,J)=I*J.
  =TYPE (FOR I=1 TO 5: (FOR J=1 TO 5:A(I,J))).
   A(1,1) = 1.0
   A(1,2) = 2.0
      .
      .
      .
   A(5,4) = 20.0
   A(5,5) = 25.0
RESULTS IN ALL ELEMENTS FOR A(1,1),...,A(5,5) BEING TYPED. THE
PARENTHESES MUST BE EVENLY MATCHED(SAME NUMBER OF LEFT ONES AS RIGHT
ONES) AND THOSE AROUND THE 'FOR' ARE REQUIRED.
:
```

```
WELL, THAT CONCLUDES YOUR 'PIL' COURSE  USING METHODS OF 'COMPUTER
ASSISTED INSTRUCTION'. I ONLY HOPE THAT YOU ENJOYED THE EXPERIENCE
AND,NOT LEAST OF ALL,LEARNED SOME 'PIL'.
NOW FOR YOUR DEMONSTRATION CLASS.


    AND SO TO YOUR THIRD EXAMPLES CLASS! I HOPE YOU'RE USING THE
REQUESTS TO YOUR FULL ADVANTAGE.
    I'VE JUST HAD A THOUGHT! YOU HAVEN'T HAD AN EXAMPLE USING ARRAYS
YET - SO LET'S KICK OFF WITH ONE.

YOU ARE GIVEN A SET OF DATA WHICH CONSISTS OF A SERIES OF AGES
IN THE RANGE 0-99 TERMINATED BY A NUMBER GREATER THAN 100. WRITE A
PROGRAM TO COUNT AND OUTPUT THE NUMBER OF AGES IN THE RANGES 0-19,20-39
,40-59,60-79 AND 80-99 USING AN ARRAY, COUNT, OF 5 ELEMENTS FOR THE
5 COUNTERS.
    NOTE THAT THERE IS A SIMPLE WAY TO DETERMINE FROM THE VALUE OF A
NUMBER THE RANGE TO WHICH IT BELONGS.
    FOR YOUR DATA USE : 29,58,87,16,45,74,3,32,61,90,19,48,999.

    YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
INFO
PIL
HELP
ANS
ans
YOU HAVEN'T ACCESSED 'PIL' YET SO HOW CAN YOU ATTEMPT AN ANSWER!!
    YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
INFO
PIL
HELP
ANS
Info
    THE PROBLEM MAY BE DIVIDED AS FOLLOWS :
1. INITIALISE THE ARRAY TO BE USED AS THE COUNTER.
2. READ IN AN AGE AND DECIDE IF IT IS A TRUE AGE.
3. IF IT IS NECESSARY, INCREMENT THE CORRESPONDING COUNTER.
4. PRINT OUT THE VALUES OF THE COUNTERS IF A TERMINATING AGE HAS BEEN
   REACHED.

    YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
INFO
PIL
HELP
ANS
pik
INVALID REQUEST. PLEASE RE-ENTER.
pil
    'PIL' IS NOW PLACED AT YOUR DISPOSAL. TO RETURN TO REQUEST MODE
SIMPLY ENTER 'MTS' TO THE INTERPRETER.
PIL( 1:01 )  Ready:
=1.1 for I=0 to 4:set count(I)=0.
=1.2 demand age.
=1.3 if age>100, go to step 1.9
=mts
    YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
INFO
PIL
HELP
ANS
```

```
help
    YOU MAY NOW REQUEST :
A DIVISION NUMBER TOGETHER WITH EITHER 'L' OR 'P'.
INFO
BACK
3
NO TYPE GIVEN WITH DIVISION   3 . PLEASE RE-ENTER REQUEST.
3 l
'PIL' MAKES THIS DIVISION OF THE PROBLEM VERY EASY. FIRSTLY, YOU
ARE ALLOWED A SUBSCRIPT WHOSE VALUE IS 0. SECONDLY, A SUBSCRIPT WHICH
HAS A FRACTION VALUE IS TRUNCATED TO FIND OUT WHICH ELEMENT IS TO BE
ACCESSED. TOGETHER WITH THE GIVEN NOTE, DO YOU NOW KNOW 'THE SIMPLE
WAY'?

    YOU MAY NOW REQUEST :
A DIVISION NUMBER TOGETHER WITH EITHER 'L' OR 'P'.
INFO
BACK
3 p
IN THE 'SET' INSTRUCTION WHICH INCREASES THE APPROPRIATE COUNTER
BY 1, WOULD YOU AGREE THAT THE APPROPRIATE SUBSCRIPT IS (AGE/20)??

    YOU MAY NOW REQUEST :
A DIVISION NUMBER TOGETHER WITH EITHER 'L' OR 'P'.
INFO
BACK
back
    YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
INFO
PIL
HELP
ANS
pil
    'PIL' IS NOW PLACED AT YOUR DISPOSAL. TO RETURN TO REQUEST MODE
SIMPLY ENTER 'MTS' TO THE INTERPRETER.
=type part 1.

    1.1     FOR I=0 TO 4:SET COUNT(I)=0.
    1.2     DEMAND AGE.
    1.3     IF AGE>100, GO TO STEP 1.9

=1.4 set count(age/20)=count(age/20)+1.
=1.5 to step 1.2
=1.9 type count.
=do part 1.
 AGE =
=29
 AGE =
=58
 AGE =
=87
 AGE =
=16
 AGE =
=45
 AGE =
=74
```

```
 AGE =
=3
 AGE =
=32
 AGE =
=61
 AGE =
=90
 AGE =
=19
 AGE =
=48
 AGE =
=999
 ERROR AT STEP 1.3: DIRECT COMMAND ONLY
=1.3 if age>100, to step 1.9
=go.
 COUNT(0) =  3.0
 COUNT(1) =  2.0
 COUNT(2) =  3.0
 COUNT(3) =  2.0
 COUNT(4) =  2.0
=mts
     YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 ans
 IN YOU ANSWER ENTER THE VALUES YOU GOT FOR THE COUNTERS IN
 ASCENDING ORDER OF RANGE, PLEASE.
 3 2 3 2 2
 EXCELLENT. PERFECTLY CORRECT!!

     YOU MAY NOW REQUEST :
 SOLN
 NEXT
 next
 A CUSTOMER'S CREDIT NUMBER IS A STRING OF 8 DECIMAL DIGITS, S.
 A NINTH DIGIT, ADDED AT THE RIGHT HAND END TO PREVENT FRAUD, IS
 OBTAINED BY FIRST ADDING THE SUM OF THE SQUARES OF THE ODD DIGITS TO
 THE SUM OF THE CUBES OF THE EVEN DIGITS. THE LAST DIGIT IN THIS SUM IS
 THE CHECK DIGIT. WRITE A PROGRAM TO READ IN S AND OUTPUT THE VALUE OF
 ITS CHECK DIGIT.
 FOR YOUR DATA USE S = "82790677"

     YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 help
     YOU MAY NOW REQUEST :
 A DIVISION NUMBER WITH EITHER 'L' OR 'P'.
 INFO
 BACK
 Info
```

```
    THE PROBLEM MAY BE DIVIDED AS FOLLOWS :
1. INITIALISE THE SUM AND READ IN S.
2. PERFORM THE SUM ON THE ODD DIGITS.
3. PERFORM THE SUM ON THE EVEN DIGITS.
4. WORK OUT AND PRINT OUT THE LAST DIGIT IN THE SUM.

    YOU MAY NOW REQUEST :
A DIVISION NUMBER WITH EITHER 'L' OR 'P'.
INFO
BACK
21
WE NEED TO LOOK AT THE ODD DIGITS OF S,I.E. DIGITS IN POSITIONS 1,3,
5 & 7. THAT SOUNDS LIKE A 'FOR' STATEMENT! AS FOR THE SUM; EACH TIME
WE ADD THE SQUARE OF THE VALUE OF THE LAST DIGIT OF THE FIRST I DIGITS,
WHERE I IS 1,3,5 OR 7. GET IT?

    YOU MAY NOW REQUEST :
A DIVISION NUMBER WITH EITHER 'L' OR 'P'.
INFO
BACK
2p
THE 'FOR' STATEMENT WILL NATURALLY BE 'BY 2:'. AS FOR THE OBJECT
STATEMENT, THE EXPRESSION TO BE ADDED TO THE OLD SUM USES THE STRING
FUNCTIONS: 'THE VALUE OF ','M $LC STRING' & 'N $FC STRING'
IN THAT ORDER.

    YOU MAY NOW REQUEST :
A DIVISION NUMBER WITH EITHER 'L' OR 'P'.
INFO
BACK
p3
THE 'FOR' STATEMENT WILL OF COURSE BE 'BY 2:'.  AS FOR THE
OBJECT STATEMENT, THE EXPRESSION TO BE ADDED TO THE OLD SUM USES THE
STRING FUNCTIONS 'THE VALUE OF ','M $LC STRING' & 'N $FC STRING'
IN THAT ORDER.

    YOU MAY NOW REQUEST :
A DIVISION NUMBER WITH EITHER 'L' OR 'P'.
INFO
BACK

1 4
WE NEED THE LAST DIGIT OF THIS SUM WHICH IS AN INTEGER. THAT'S IT!
WHY DON'T WE TAKE THE FRACTION PART OF A TENTH OF THE SUM THEN
MULTIPLY IT BY 10?

    YOU MAY NOW REQUEST :
A DIVISION NUMBER WITH EITHER 'L' OR 'P'.
INFO
BACK
```

```
back
     YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 pil
     'PIL' IS NOW PLACED AT YOUR DISPOSAL. TO RETURN TO REQUEST MODE
 SIMPLY ENTER 'MTS' TO THE INTERPRETER.
 PIL( 1:01 )  Ready:
=1.1 set sum=0.
=1.2 demand s.
=1.3 for i=1 to 7 by 2:set sum=sum+(the value of (1 $ic i $fc s))**2.
=1.4 for i=2 by
              to 8 by 2:set sum=sum+(the value of (1 $ic i $fc s))**3.
=1.5 set x=fp of sum/10*10.
=1.6 set_

LINE DELETED
=1.6 type x.
=do part 1
 S =
="82790677"
 X =  0.0
=mts
     YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 ans
 IN YOUR ANSWER PLEASE ENTER THE CHECK DIGIT.
 0
 INCORRECT.
 AS THAT WAS YOUR FIRST ATTEMPT I SUGGEST THAT YOU RETURN TO REQUEST
 MODE AND TYPE 'HELP'.

     YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 help
     YOU MAY NOW REQUEST :
 A DIVISION NUMBER WITH EITHER 'L' OR 'P'.
 INFO
 BACK
 p5
 INVALID REQUEST. PLEASE RE-ENTER.
 p4
 HEARD OF THE ARITHMETIC FUNCTION 'FP OF '??

     YOU MAY NOW REQUEST :
 A DIVISION NUMBER WITH EITHER 'L' OR 'P'.
 INFO
 BACK
 back
```

```
    YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 pil
    'PIL' IS NOW PLACED AT YOUR DISPOSAL. TO RETURN TO REQUEST MODE
 SIMPLY ENTER 'MTS' TO THE INTERPRETER.
=1.5 set x=fp of (sum/10*10).
=do part 1.
 S =
="82790677"
 X =  0.0
=1.5 set x=fp of (sum/10)*10.
=do part 1.
 S =
="82790677"
 X =  8.0
=mts
    YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 ans
 IN YOUR ANSWER PLEASE ENTER THE CHECK DIGIT.
 8
 EXCELLENT. PERFECTLY CORRECT!!

    YOU MAY NOW REQUEST :
 SOLN
 NEXT
 next
 WRITE A PROGRAM TO READ IN A CHARACTER STRING, S, AND PRINT OUT
 THE NUMBER OF LETTERS, L, AND THE NUMBER OF DIGITS, D, CONTAINED IN
 THE STRING.
 FOR YOUR DATA USE S="X?P37BG**P9P8P7?+Q-X3*X4.7"

    YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 info
    THE PROBLEM MAY BE DIVIDED AS FOLLOWS :
 1. INITIALISE THE COUNTERS AND READ IN S.
 2. SET UP A SUITABLE LOOP.
 3. INCREMENT THE COUNTERS ACCORDING TO EACH CHARACTER.
 4. PRINT OUT THE RESULTS.

    YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 pil
    'PIL' IS NOW PLACED AT YOUR DISPOSAL. TO RETURN TO REQUEST MODE
 SIMPLY ENTER 'MTS' TO THE INTERPRETER.
 PIL( 1:01 )  Ready:
```

```
=1.1 set l=0
=1.2 set d=0
=1.3 demand s.
=1.4 set len=l of s.
=1.5 for l=1 to len: do part 2.
=2.1 set r=1 $lc l $fc s.
=2.2 if r $ge "0",set d=d+1;set l=l+1.
=do part 1.
 S =
="x?p37bg**p9p8p7?+q-x4.7"
=type l,d.
 L =  16.0
 D =  7.0
=mts
     YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 ans
 IN YOUR ANSWER PLEASE ENTER THE NUMBER OF LETTERS AND THE NUMBER
 OF DIGITS IN THAT ORDER.
 16  7
 INCORRECT.
 AS THAT WAS YOUR FIRST ATTEMPT I SUGGEST THAT YOU RETURN TO REQUEST
 MODE AND TYPE 'HELP'.

     YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 pil
     'PIL' IS NOW PLACED AT YOUR DISPOSAL. TO RETURN TO REQUEST MODE
 SIMPLY ENTER 'MTS' TO THE INTERPRETER.
=do part 1.
 S =
="x?p37bg**p9p8p7?+q-x3*x4.7"
=type l,d.
 L =  18.0
 D =  8.0
=mts
     YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 ans
 IN YOUR ANSWER PLEASE ENTER THE NUMBER OF LETTERS AND THE NUMBER
 OF DIGITS IN THAT ORDER.
 18  8
 SORRY, BUT THAT IS THE WRONG ANSWER.
     YOU MAY REQUEST ONE OF :
 BACK
 EXPL
 expl
```

EXPLANATION.

THE STRING IS READ IN AND L & D SET TO 0. THE LENGTH OF THE INPUT STRING IS FOUND AS IT IS THIS NUMBER OF CHARACTERS WE WILL BE COMPARING IN THE PROBLEM. A LOOP IS THEREFORE SET UP USING A 'FOR' STATEMENT WITH THE LENGTH OF S AS THE LIMIT. THE OBJECT STATEMENT OF THIS 'FOR' IS BEST A 'DO PART N.' INSTRUCTION, NOT ONLY FOR CLARITY BUT ALSO FOR A SLIGHT SAVING OF WORK. AS WE MAKE 2 SETS OF COMPARISONS ON THE ISOLATED CHARACTER, IT IS PERHAPS BEST TO ASSIGN THIS CHARACTER TO A STRING VARIABLE R HAVING WORKED IT OUT ONCE ONLY. TO WORK IT OUT, THE 'LAST CHARACTER OF THE FIRST I CHARACTERS OF ' IS USED. THE COMPARISONS ARE AS FOLLOWS: FOR DIGITS, IF R IS GREATER THAN OR EQUAL TO "0", D IS INCREMENTED; FOR LETTERS, IF R IS GREATER THAN OR EQUAL TO "A" AND LESS THAN OR EQUAL TO "Z", L IS INCREMENTED. ON COMPLETION OF THE LOOP, L AND D ARE OUTPUT.
:

SPECIMEN PROGRAM.

```
  =3.1 DEMAND S.
  =3.2 SET LENGTH=L OF S.
  =3.3 SET L=0.
  =3.4 SET D=0.
  =3.5 FOR I=1 TO LENGTH: DO PART 4.
  =3.6 TYPE L,D.
  =4.1 SET R=1 $LC I $FC S.
  =4.2 IF R $GE "0",SET D=D+1.
  =4.3 IF R $GE "A" $AND R $LE "Z",SET L=L+1.
FOR S="X?P37BG**P9P8P7?+Q-X4.7", L=10 AND D=8.
```

USING THE PROGRAM YOU ALREADY HAVE FOR THE SUMMATION OF THE G.P.:

$$A , AR , AR^{2} ,\ldots\ldots, AR^{N-1}$$

AMEND IT TO GIVE OUT THE VALUES OF :
A) THE N TH. TERM IN SCIENTIFIC NOTATION WITH 5 PLACES OF DECIMALS.
B) THE SUM IN STANDARD NOTATION WITH 3 PLACES OF DECIMALS.
FOR YOUR DATA USE A=12, R=0.5 AND N=20.

YOUR REQUEST MAY BE ONE OF THE FOLLOWING :

```
INFO
PIL
HELP
ANS
Info
```

THE PROBLEM MAY BE DIVIDED AS FOLLOWS :
1. A 'FORM' FOR THE N TH. TERM.
2. A 'FORM' FOR THE SUM.

```
    YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 pil
     'PIL' IS NOW PLACED AT YOUR DISPOSAL. TO RETURN TO REQUEST MODE
 SIMPLY ENTER 'MTS' TO THE INTERPRETER.
 PIL( 1:01 )  Ready:
=form 1.
=.....
=form 2.
=!_!_!_.!_!_!_
=1.1 demand a,r,n.
=1.2 set term=a.
=1.3 set sum=a.
=1.4 for i=1 to n-1:do part 2.
=1.5 type in form 1,term.
=1.6 type in form 2,sum.
=2.1 set term=term*r.
=2.2 set sum=sum+term.
=do part 1.
 A =
=12
 R =
=0.5
 N =
=20
  2.2888E-05
  23.999
=mts
     YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 help
     YOU MAY NOW REQUEST:
 A DIVISION NUMBER TOGETHER WITH EITHER 'L' OR 'P'.
 INFO
 BACK
 1 p
 QUITE SIMPLY, SCIENTIFIC NOTATION USES 1 PERIOD PER CHARACTER OF
 WIDTH.

     YOU MAY NOW REQUEST:
 A DIVISION NUMBER TOGETHER WITH EITHER 'L' OR 'P'.
 INFO
 BACK
 1 1
 FOR 5 PLACES OF DECIMALS, SCIENTIFIC NOTATION NEEDS A WIDTH OF
 (5+7).

     YOU MAY NOW REQUEST:
 A DIVISION NUMBER TOGETHER WITH EITHER 'L' OR 'P'.
 INFO
 BACK
 back
```

```
    YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 pil
    'PIL' IS NOW PLACED AT YOUR DISPOSAL. TO RETURN TO REQUEST MODE
 SIMPLY ENTER 'MTS' TO THE INTERPRETER.
=type form 1.
 ..........
=form 1.
=...........
=type form1,foem 2.
 Eh?  FORM1  =?
=type form 1,form 2.
 ...........
 ___.___
=do part 1.
 A =
=12
 R =
=0.5
 N =
=20
  2.28881E-05
  23.999
=mts
    YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 ans
 IN YOUR ANSWER, PLEASE ENTER YOUR VALUES FOR THE N TH. TERM AND THE
 SUM IN THAT ORDER. BE SURE TO ENTER THEM IN THE EXACT FORMAT OF YOUR
 OUTPUT.
  2.2888e-05,23.999
 INCORRECT.
 AS THAT WAS YOUR FIRST ATTEMPT I SUGGEST THAT YOU RETURN TO REQUEST
 MODE AND TYPE 'HELP'.

    YOUR REQUEST MAY BE ONE OF THE FOLLOWING :
 INFO
 PIL
 HELP
 ANS
 ans
 IN YOUR ANSWER, PLEASE ENTER YOUR VALUES FOR THE N TH. TERM AND THE
 SUM IN THAT ORDER. BE SURE TO ENTER THEM IN THE EXACT FORMAT OF YOUR
 OUTPUT.
  2.28881e-05,23.999
```

```
EXCELLENT. PERFECTLY CORRECT!!

    YOU MAY NOW REQUEST :
SOLN
NEXT
soln
    SPECIMEN PROGRAM.

  =FORM 1.
  =.............
  =FORM 2.
  =!_!_.!_!_!_
  =TYPE FORM 2.

   __.___
  =5.1 DEMAND A,R,N.
  =5.2 SET TERM=A.
  =5.3 SET SUM=A.
  =5.4 FOR I=1 TO N-1:DO PART 6.
  =5.5 TYPE IN FORM 1,TERM.
  =5.6 TYPE IN FORM 2,SUM.
  =6.1 SET TERM=TERM*R.
  =6.2 SET SUM=SUM+TERM.

TEN MINUTES TO SHUTDOWN SO PLEASE PREPARE TO SIGNOFF        THANK YOU     OPER

    THAT'S YER PROVERBIAL LOT, MATE! I HOPE THAT YOU FOUND YOUR
EXAMPLES CLASSES ON THE COMPUTER AT LEAST INTERESTING IF NOTHING ELSE.
AND, OF COURSE, I HOPE YOU KNOW ALL ABOUT 'PIL' NOW!!
TOTAL TIME=  1.98 HRS. THIS SESSION=118.78 MINS.
```

APPENDIX F.

The student performance information corresponding to that session.

CURRENT POSITION: LESSON 6,SEGMENT 5,CODE 7333

LESSON	QUESTION	A,U OR N	#CA & #WA VALUES	TIME	UNANTICIPATED REPLY
3	1	A	1 0 0 0 0 0 0 0 0 0-1 0 0 0 0 0 0 0 0 0 0	18	
3	2	U		53	NO IDEA!
3	2	A	-1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0	12	
3	4	A	1 0 0 0 0 0 0 0 0 0 0-1 0 0 0 0 0 0 0 0 0 0	27	
3	5	A	1 0 0 0 0 0 0 0 0 0 0-1 0 0 0 0 0 0 0 0 0 0	19	
3	6	U		62	ABBC
3	6	A	-1 1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	16	
3	6	A	1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	16	
3	7	U		74	HELP??
3	8	A	-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	67	
3	8	A	-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	26	
3	8	A	-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	20	
3	8	A	1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	60	
3	7	A	1 0 0 0 0 0 0 0 0 0 0-1-1 0 0 0 0 0 0 0 0 0	11	
3	9	U		42	DON'T KNOW
3	10	A	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	26	
3	9	U		47	5
3	9	A	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	21	
3	11	A	1 0 0 0 0 0 0 0 0 0 0-1 0 0 0 0 0 0 0 0 0	12	
3	13	A	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	23	
3	15	A	-1 0 0 0 0 0 0 0 0 0 0 1-1 0 0 0 0 0 0 0 0	39	
3	16	U		38	" 4.0 "
3	16	A	1 0 0 0 0 0 0 0 0 0 0-1 0 0 0 0 0 0 0 0 0	34	
3	17	A	1 0 0 0 0 0 0 0 0 0 0-1 0 0 0 0 0 0 0 0 0	13	
3	18	A	1 0 0 0 0 0 0 0 0 0 0-1-1 0 0 0 0 0 0 0 0	24	
3	19	A	-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	27	
3	19	A	-1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	36	
3	20	A	1 0 0 0 0 0 0 0 0 0 0-1 0 0 0 0 0 0 0 0 0	19	
3	21	A	-1 1 0 0 0 0 0 0 0 0 0-1-1 0 0 0 0 0 0 0 0	35	
3	22	A	-1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0	11	
3	24	A	1 0 0 0 0 0 0 0 0 0 0-1 0 0 0 0 0 0 0 0 0	21	
7	27	A	-1 0 0 0 0 0 0 0 0 0 0 1-1-1 0 0 0 0 0 0 0	29	
7	27	A	1 0 0 0 0 0 0 0 0 0 0 0-1-1-1 0 0 0 0 0 0 0	33	
7	28	A	1 0 0 0 0 0 0 0 0 0 0 0-1-1 0 0 0 0 0 0 0 0	17	
7	29	A	-1 0 0 0 0 0 0 0 0 0 0 0-1 1 1 0 0 0 0 0 0	30	
7	29	A	1 0 0 0 0 0 0 0 0 0 0 1-1-1 0 0 0 0 0 0 0	11	
7	30	A	-1 0 0 0 0 0 0 0 0 0 0 0-1 1 0 0 0 0 0 0 0	64	
7	30	U		17	1.23456E+04
7	31	A	1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0	25	
7	32	A	1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0	17	
7	33	A	1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0	14	
6	1	A	-1-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	29	

```
6          1        A       1-1-1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  39
6          1        U                                                    19:PIK
6          1        A      -1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   6
6          1        A      -1-1 1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   9
6         13        A       0-1-1 1-1 0-1-1-1-1 0 0 0 0 0 0 0 0 0 0 0 0   9
6         13        A       0-1-1 1-1 0 1-1-1-1 0 0 0 0 0 0 0 0 0 0 0 0   8
6         13        A       0-1-1 1-1 0-1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0   4
6         13        A       0-1-1-1-1 0-1-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0   4
6          1        A      -1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   5
6          1        A      -1-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   4
6         14        A       1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  15
6          2        A      -1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   5
6          1        A      -1-1 1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   7
6         23        A       0-1-1-1-1 0-1-1 1-1 0 0 0 0 0 0 0 0 0 0 0 0  19
6         23        A       0-1 1-1-1 0 1-1-1-1 0 0 0 0 0 0 0 0 0 0 0 0  25
6         23        A       0-1 1-1-1 0-1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0   5
6         23        A       0-1-1 1-1 0-1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0   3
6         23        A       0-1-1-1 1 0 1-1-1-1 0 0 0 0 0 0 0 0 0 0 0 0  12
6         23        A       0-1-1-1-1 0-1-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0   7
6          1        A      -1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   3
6          1        A      -1-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   4
6         24        U                                                     6:0
6          1        A      -1-1 1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  14
6         23        A       0-1-1-1-1 0-1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0   4
6         23        A       0-1-1-1 1 0-1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0   8
6         23        A       0-1-1-1-1 0-1-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0  13
6          1        A      -1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   2
6          1        A      -1-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   3
6         24        A       1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   7
6          2        A      -1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   8
6          1        A       1-1-1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  17
6          1        A      -1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  24
6          1        A      -1-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   5
6         34        U                                                     9:16  7
6          1        A      -1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   2
6          1        A      -1-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   7
6         34        U                                                     5:18  8
6          3        A      -1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   8
6          1        A       1-1-1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   4
6          1        A      -1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  75

6          1        A      -1-1 1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   4
6         43        A       0 1-1 0 0 0-1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0  10
6         43        A       0 1-1 0 0 0 1-1-1-1 0 0 0 0 0 0 0 0 0 0 0 0   6
6         43        A       0-1-1 0 0 0-1-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0   4
6          1        A      -1 1-1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   2
6          1        A      -1-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  20
6         44        U                                                     19: 2.2888E-05,23.999
6          1        A      -1-1-1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   6
6         44        A       1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  17
6          2        A       1-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   6
TOTAL TIME= 119 MINS. LAST SESSION= 119 MINS.
```