4934

NATIONAL LIBRARY

OTTAWA

BIBLIOTHÈQUE NATIONALE

OTTAWA

NAME OF AUTHOR..DAVID....PAUL...FLATHMAN.....

TITLE OF THESIS..LIST..PROCESSING..SIMULATION

..OF...COMPUTER-ASSISTED.....

....INSTRUCTION.....

UNIVERSITY....UNIVERSITY...OF..ALBERTA.....

DEGREE..Ph.D.....YEAR GRANTED....1969.....

Permission is hereby granted to THE NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed). David P. Flathman

PERMANENT ADDRESS:

#18-8805-111 St.

Edmonton.....

...Alberta.....

DATED. October 9 1969

NL-91

THE UNIVERSITY OF ALBERTA

LIST PROCESSING SIMULATION

OF

COMPUTER-ASSISTED INSTRUCTION

BY

© DAVID PAUL FLATHMAN

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF EDUCATIONAL PSYCHOLOGY

EDMONTON, ALBERTA

FALL, 1969

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES

The undersigned hereby certify that they have read and recommended to the Faculty of Graduate Studies for acceptance, a thesis entitled, "List Processing Simulation of Computer-Assisted Instruction" submitted by David Paul Flathman in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

S. Hampton

Supervisor

W.S.A. Adam

J.O. Maguire

Robert S. Ware

Harold E. Mitzel

External Examiner

Date: October 6, 1969

ABSTRACT

The purpose of this thesis was to simulate the interaction between a student and the IBM 1500 instructional computer system.

A set of conventions and codes was established to develop a list structural representation of computer-assisted instructional sequences written in the language Coursewriter II for the 1500. This representation was then programmed for the IBM 360/67 computer. Accurate representations were produced for a variety of courses.

Three areas of application of the list structural representation were investigated. The first product was a two-dimensional logic chart which diagrams the complete instructional logic of a course. Sequential execution, branches, and merges are indicated on the chart. The second application was to trace all possible paths that could be followed by students making consistent responses through a course. These applications provide needed assistance for authors in the development and documentation of courses.

The third application was the production of a framework for the simulation of student response models. The framework requires the specification of the condition for branching at each response-dependent decision point in the course. Student simulation variables depending on the path may be constructed. This application provides a mechanism for testing student learning models.

ACKNOWLEDGEMENTS

The writer wishes to express his sincere thanks to the supervisor of the thesis, Dr. S.M. Hunka, for his advice and guidance during the period of the study. Thanks are also expressed to the committee members, Prof. W.S. Adams and Dr. T.O. Maquire.

The writer wishes to acknowledge with gratitude the financial assistance received from the University of Alberta.

Appreciation is also expressed to the Social Science Research Council for a grant for intensive study of computer simulation programs. This provided an opportunity for the writer to receive the assistance of Dr. A. Newell and Dr. H.A. Simon at the Carnegie Institute of Technology.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

Introduction

INTRODUCTION

A. Computer-Assisted Instruction

The digital computer has begun to be used experimentally as a medium for the instructional process in a wide variety of subject matter areas. Computer-assisted instruction (CAI) originated with the work of Rath and his colleagues (Rath, Anderson, and Brainerd, 1959) and was quickly taken up by other research groups (Coulson, 1962). Literature in the area of CAI has been surveyed recently by several authors, including Zinn (1967), Hickey (1968), and Silberman and Filep (1968).

CAI refers to the interactive instructional and learning process between computer and student. The computer stores stimulus material, presents it to the student, accepts and analyzes student responses, and records student performance. The student examines the stimulus material, responds, and waits for the computer's reply.

The needs of students in the interactive CAI situation impose certain requirements upon the computer system. A large memory capacity is needed to store all the curriculum or stimulus material, as well as records of student performance. Reasonably fast response time is desirable to avoid student inattention, and some form of time-sharing is required in order that different students may work on different subject matter at the same time. Interactive media for stimulus and response should be available in a variety of sensory modes, particularly auditory and visual modes. This may involve films, graphics, audio tapes, and the like.

Another important requirement of a CAI system is software, particularly an author language that permits convenient entry of stimulus materials and instructional logic into the system.

A recent development in CAI is the commercial marketing of instructional computer systems for research purposes. For example, firms that have specific thrust in this area include RCA, Honeywell, Philco-Ford, and IBM. One such experimental system is the University of Alberta IBM 1500 Instructional System.

With the 1500 system, curriculum material and instructional sequences are stored in peripheral memory banks (disks), while the central processing unit presents the material to each student and analyzes student responses. The equipment at each student terminal consists of a typewriter or combination display screen and keyboard, a light pen, and a film strip projector. As well, an audio play and record unit is being made available for each student terminal at the time of writing.

Curriculum or stimulus material may be presented visually to the student by means of the typewriter or screen and the projector. Characters or graphics of any design, within physical limitations, may be presented on the display screen cathode ray tube (IBM 1500 Coursewriter II Author's Guide, 1967). Film strip slides may be shown individually in any order. Student responses constructed from characters are accepted from the typewriter or keyboard. As well, the system can recognize a lighted area of the screen that is pointed to with the light pen. The system is "time-sharing" in the sense that each user at a terminal may be working on different course material at the same time. The system switches automatically between terminals in such a way that each user

need only be conscious of his own interaction with the computer.

System users may conveniently be categorized into three types: proctors, authors, and students. Proctors operate the system, schedule courses, register students, and the like. Authors have the responsibility of developing curriculum material and instructional sequences. This includes the programming of such material in a language acceptable to the computer, for example, the language Coursewriter II for the 1500 system. Students interact with and respond to the course material prepared by authors. Each category of user has a set of system control commands at his disposal. For example, author commands include those that permit the insertion, deletion, replacement, movement, display, and execution of Coursewriter statements in a course.

Computer-assisted instruction provides a medium in which controlled experimentation on the instructional and learning processes can be made. The extent to which CAI may have practical application in actual schools is not yet fully known.

B. The Need for IBM 1500 Author Assistance

Programming a course for CAI on the 1500 computer system is a very large chore. Besides requiring familiarity with curriculum material and instructional methodology, the author must possess a working knowledge of the Coursewriter II language. In contrast to algebraic languages such as Fortran, Coursewriter II provides relatively few diagnostic messages to help the programmer during assembly or execution. Instructional programs tend to be long and logically complex, and the actual paths followed by students through a program depend on the responses they make. The logical

complexity of Coursewriter II programs is greatly increased by a feature of the language that provides for automatic or implicit branching. For example, using the path-tracing program which is one product of this thesis, more than 200 distinct paths were found in a course consisting of only about 500 Coursewriter II statements.

At present it is usually the case that the only precise representation of the logic of an instructional sequence is the CAI program itself. Usually this has been developed from existing book-form curriculum material at the author's initiative, perhaps with the aid of flowcharts. Flowcharts may be used to describe course logic at various levels, depending on the amount of detail and precision required. At the most detailed level, a flowchart may be a precise representation of programming logic, and one that is readily comprehensible once the conventions are understood. At another level, flowchart blocks may represent logical units greater than a single instruction, hence their use in this way may be symbolic and less precise.

Detailed flowcharts are generally not available for any CAI program, as their construction by hand is too time-consuming.

It is generally feasible to completely debug an algebraic program by testing all possible contingencies that may arise. Such a procedure is more time-consuming with instructional programs on account of their length, logical complexity, and response-dependency. All possible student responses must be considered in order to fully debug a CAI course. To do this, an author would have to sign on as a student and try out all responses that could be made in order to be assured that the program takes appropriate action for each possible response.

In order to implement an instructional program, the author needs to have an idea about the responses actual students will make, the difficulty of the material, and time estimates. Grubb (1967, p.71) describes his personal experience in writing a computerized course: "In writing the statistics course for this medium I found that I was revising the course almost hourly. Student-record data from the computer on students' performance would almost always open up in the course material new branch structures that I was not clever enough to predict."

A further burden for the CAI author is the task of documentation of a completed course. Exactly how much detail need be included in a documentation depends on for whom it is being prepared, but without a form of documentation a program is not very useful to others. All the aforementioned reasons for difficulties in debugging apply also to documentation, making it also very time-consuming, expensive, and inefficient.

C. Computer Simulation

A computer simulation of an operation is a process of modelling the operation by representing it with a computer program. The operations of numerous industrial and administrative systems have been effectively and economically modelled through the use of computer simulation techniques. New computer languages, both general and specific in their application, have been developed to meet the needs of those charged with the programming of computer simulation models. One of the main purposes of such work has been to optimize the parameters of a complex operation under varying conditions. Naylor, Balintfy, Burdick, and Chu (1966) provide a summary of simulation techniques in economics.

Digital simulation has been used to advantage in behavioral science as well. Computer models of cognitive processes have been particularly promising, and are closely paralleled by numerous endeavours in the area of artificial intelligence (Feigenbaum and Feldman, 1965). A computer simulation of student performance in a computer-controlled instructional setting is not yet reported in the literature.

The point of view developed in this thesis is to demonstrate how computer simulation techniques can be applied to CAI in such a way as to provide needed assistance for authors in the development of instructional programs. It is also shown that computer simulation of CAI may be used to test student learning models.

The idea of simulating CAI is to provide a representation of both the instructional logic and possible student responses so that information of benefit to authors may result. For example, a flowchart of the instructional logic could be drawn from the simulation representation. As well, the paths followed by different types of students through an instructional sequence could be traced, to give an idea of how a variety of students might respond to the CAI course. Information derived from the simulation could be used to develop, debug, improve, and document the instructional program.

CHAPTER II

Background and Methodology

BACKGROUND AND METHODOLOGY

A summary of the computer language Coursewriter II is given in section A of this chapter. A comparison of the advantages and disadvantages of three alternative approaches to simulation of the instructional and learning aspects of CAI is made in section B. Of the three suggested possible methods, the last one, simulation using a list-processing language, is selected. A brief description of a modified symmetric list processor (MOSLIP) is then given in section C.

A. Summary of Coursewriter II

Instructional programs for the IBM 1500 are written in the computer language Coursewriter II. The programmer or author of a computerized course writes the sequence of instructions as if for one student, although in fact a number of students may be at different points in the same course, or another course, at the same time, because of the time-sharing nature of the system. Each student station consists of a film strip projector, audio equipment, and a typewriter or display screen with keyboard and light pen. For each station, areas in the computer memory are set aside as buffers, counters, and switches to keep track of student responses, to compute scores, and to provide for conditional branches. Buffers, counters, and switches are used to store character string, integer, and logical data respectively.

Details of the Coursewriter II language are given in the IBM 1500 Coursewriter II Author's Guide (1967). The thirty-five instructions of the language may be summarized here by six main areas.

The summary gives the two-character operation code (card columns 7 and 8) which calls up each instruction, followed by a brief explanation of the operation performed by each instruction:

1. Problem presentation

pr	problem start (begin new problem)
ty	type text or contents of buffer on typewriter
dt	display text or contents of buffer on display screen
dg	display graphic on display screen
de	erase one or more lines on the display screen
dl	display emphasis line (underline) on display screen
pm	proctor message sent to proctor station
au	position and/or play or record audio message
fp	position film and/or open or close shutter
pa	pause

2. Response request

ep	enter and process response from keyboard or light pen
ec	enter response and continue

3. Response analysis (compare stored answer with student's response)

ca	correct answer
cb	synonymous correct answer
wa	wrong answer
wb	synonymous wrong answer
aa	additional answer
ab	synonymous additional answer
un	unrecognized response
ea	end of answers

4. Scorekeeping

ad add integer or counter to counter

sb subtract integer or counter from counter

mp multiply counter by integer or counter

dv divide counter by integer or counter

ld load integer or counter into counter, load switch, or load text or buffer into buffer

5. Presentation sequence control

nx no execute (conditional execution of subsequent instructions)

tr transfer to new course segment

br branch to label, return register, last executed ep, or n'th next problem, either unconditionally or conditional on the state of a counter or switch

lr load label into return register

6. Special instructions

cm call and execute a Coursewriter macro

fn call and execute an assembly-language function

no no operation

ma macro name (beginning of macro)

em end of macro

en end of course

Most of these instructions may have modifiers and/or parameters which specify the details of the operation to be performed.

In addition to explicit branches which may be constructed by the use of statement labels and the br instruction, Coursewriter II also features

implicit or automatic branching. The entire logic of the implicit branching feature of the language is summarized in the course flow decision table (Figure 2.1) reprinted here from the IBM 1500 Coursewriter II Author's Guide (1967).

All Coursewriter II instructions may be classified as either major or minor. The major instructions include those previously classified as response analysis instructions (category 3 above) as well as the pr and nx instructions. Whenever a control instruction (a major or ep) is executed, a condition is set up whereby subsequent instructions may or may not be executed. The nine conditions that may be set up are indicated in columns 0 through 8 of the course flow decision table, while the eight types of instructions that may be currently encountered are indicated as rows 0 through 7 of the table. The decision to be taken when an instruction is encountered under a given condition is indicated in the body of the course flow decision table at the intersection of the appropriate row and column.

For example, minor instructions (row 4 of Figure 2.1) following an executed ca or cb (correct answer) are executed or not depending on whether the student's response did or did not match the stored correct answer (columns 7 or 3 respectively). As another example, when an aa, ca, wa, or un (row 2) is encountered following a match on a ca or cb (column 7), course flow automatically skips to the next problem (pr).

Implicit branching introduces a hidden, but useful, complexity in the logic of instructional programs written in Coursewriter II.

LAST EXECUTED CONTROL INSTRUCTION AND/OR CURRENT CONDITION

CURRENTLY ENCOUNTERED INSTRUCTION

	0. un (counter)* Mismatch	1. ep Timed Out	2. aa,ab Mismatch	3. ca,cb Mismatch	4. wa,wb Mismatch	5. pr nx ea	6. aa,ab Match ep in Time	7. ca,cb Match	8. wa,wb,un (counter)* Match
0. ea or end-of-course	No Execute RETURN TO un (8)	No Execute SKIP TO pr (5)							No Execute RETURN TO LAST Executed ep (1 or 6)
1. nx	Execute (5)						No Execute SKIP TO NEXT Major		
2. aa	Execute (2 or 6)								
2. ca	Execute (3 or 7)								
2. wa	Execute (4 or 8)								
2. un	Execute (0 or 8)								
3. pr	No Execute RETURN TO un (8)	Execute (5)							
4. minor	No Execute EXAMINE NEXT INSTRUCTION (no change)					Execute tr (5) ep (1,6) Other (no change)			
5. wb					Execute (4 or 8)	No Execute EXAMINE NEXT INSTRUCTION (no change)			
6. cb				Execute (3 or 7)					
7. ab			Execute (2 or 6)						

Major Instructions
ea,nx,aa,ca,wa,un, pr,ab,cb,wb

Minor Instructions
ec,ep,br,tr,ld,lr,ad,sb,mp,dv,fn,dt, dg,dl,de,au,fp,ty,pa,pm,no,label

*Counters refer to UNI and UN2, which are an integral part of un instruction logic

Figure 2.1

Course Flow Decision Table

B. Alternative Methods of Simulation

The intent of the thesis was stated in Chapter I as the development of means for computer simulation of student performance in a computer-assisted instructional setting. Input to the proposed simulation program would consist of a course program (written in Coursewriter II for the IBM 1500) and required parameters would include debug, documentation, and learning options depending on the principal intent of the simulation. The learning parameters would determine, on a deterministic or probabilistic basis, the decision to be taken at each response-dependent branch point of the course. Output would include a trace of course flow, and particular debug and documentation information requested.

Three alternative methods are initially considered, depending on the type of simulation language used. The three types of languages discussed here are Coursewriter II, a general purpose simulation language, and a list-processing language as the medium for simulation.

1. Simulation using Coursewriter II

Since the input course program in its existing form already represents a model of the instructional process for the course, only simulated student responses (or their branching equivalents) need be added for the program to represent the complete learning and instructional process. Thus the simulated program would be written in the same language as the input course program -- Coursewriter II.

Since it could be run on the same computer (the IBM 1500), the simulation program could easily replicate the details of the course program (for example, graphic displays). This feature would make this

method particularly useful in debugging. Unfortunately, there are a number of disadvantages to this approach:

(a) Simulation would tend to be slow, and might use up valuable author or student time.

(b) It would be difficult to provide a clear separation of learning aspects from instructional aspects.

(c) The inflexible nature of the Coursewriter language would tend to make implementation difficult.

d) The representation would not be useful for documentation or flowcharting, since it provides no alternative representation of course logic other than that contained in the course program itself.

2. Simulation using a general purpose simulation language

The language selected must be flexible and suited to the special needs of simulation, such as IBM's General Purpose Systems Simulator (GPSS). GPSS has been used to simulate vehicle traffic, factory production lines, communication systems and the like, and could be used to represent student flow through a CAI course. The starting point of a GPSS program is a block diagram similar to a flowchart with precise conventions. Possible output could include the number of students passing through any point in the system, time required for completion of portions of the course, average utilization of system elements and queue lengths at selected points.

GPSS might be more useful for study of performance of the whole 1500 system, rather than just the portion consisting of student interaction with a course program. The main disadvantage of GPSS is that the starting point

(a block diagram) does not meet the needs of the proposed simulation which is required to begin with an existing CAI course program, for which no flowchart is available. A typical Coursewriter II program contains a large number of system elements, each of which would have to be represented by a block diagram.

Thus GPSS, although potentially useful in a study of the 1500 system, is not well suited to the needs of the proposed simulation.

3. Simulation using a list processing language

List processing languages have been found useful with a variety of complex symbol-manipulation problems, including language translation and artificial intelligence. Hierarchically organized data of indefinite length is stored in cells linked together to form lists. Lists in turn may be linked together by the provision that a list cell may point to a sublist. A tree structure results if sublists may be pointed to only once. A more complex list structure results if sublists may be sublists of more than one list, as is permitted in the more general list-processing languages. A detailed comparison of four well-known list-processing languages (LISP, IPL-5, COMIT, and SLIP) is given by Bobrow and Raphael (1964).

SLIP is a symmetric list processor consisting primarily of a set of Fortran subprograms written by Weizenbaum (1963). A modified symmetric list processor (MOSLIP) designed for the IBM 360/67 computer is described by Flathman (1968).

Using MOSLIP, instructional logic would be represented isomorphically as a data structure that could be operated on in a variety of ways.

Details of an instructional program, such as graphical displays, could not be represented in their original form during a simulation so that certain types of debug errors could not be located. However, the instructional logic and branching mechanism could be completely represented. Powerful list operations are provided in the language, and the well-known algebraic flexibility of Fortran is also available. Learning and instructional aspects of a CAI simulation could be clearly separated. Thus a list-processing language such as MOSLIP would seem to be the most advantageous for the proposed application to CAI simulation.

The first major problem to be solved is to find a convenient and economical representation of all Coursewriter instructions affecting branching, both explicit and implicit, so that the logic of a complete course could be represented as a data list structure. Then, operating on this data base, a variety of debug, documentation and learning simulation programs could be written. Chapters III and IV of this thesis are devoted to the solution of these problems.

To illustrate how list-processing may be applied to representing program logic, an outline of a short but typical sequence of Coursewriter instructions is given:

ep request response

dt display text if response is in time

nx skip to ca if response is in time

de display erase if response is timed-out

ca compare with stored correct answer

If the student responds in time to the response request ep instruction, then it may be seen from the course flow decision table (Figure 2.1) that

the dt and then the ca instructions are executed. In this case the nx and de are not executed. However, if the student does not respond within the permitted latency time, then the ep is timed out and dt is not executed, but instead the nx, de, and ca are executed.

A possible MOSLIP list structure representation of the program logic in this short sequence of Coursewriter instructions is given in Figure 2.2.

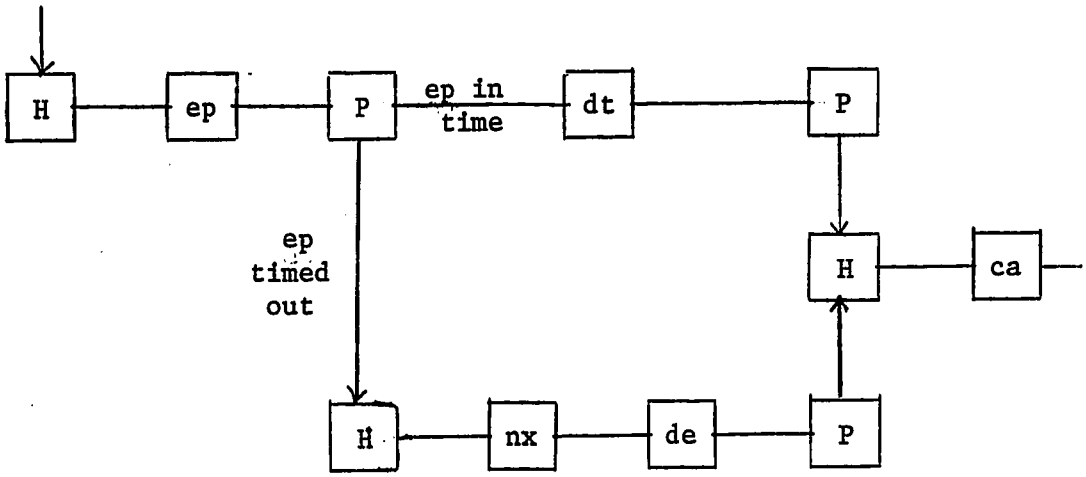

Figure 2.2 A MOSLIP Representation

In Figure 2.2, each box represents a MOSLIP cell. A box with an H in it represents the header of a list, while a box with a P is a cell pointing to the header of a sublist. Cells on a list are indicated by horizontal lines from the list header on the left to the end of the list on the right. The occurrence of sublists is indicated by the vertical arrows.

There are three lists in this representation, one for each control instruction (major instruction or ep). The control instructions in this example are the ep, nx, and ca, and there is a list header (H) for each. The two paths through the sequence, (ep dt ca) and (ep nx de ca) are clearly evident in this representation.

The complete details of how a MOSLIP representation may be established for the logic of any Coursewriter course are given in Chapter III.

C. Modified Symmetric List Processor

This section comprises a brief description of the modified symmetric list processor (MOSLIP) designed by Flathman (1968). MOSLIP is based on the SLIP language invented by Weizenbaum (1963) and consists of a set of Fortran subprograms for the IBM 360/67.

In MOSLIP, information is stored in specially designed cells which also hold data identifying the cell and describing its linkage to other cells. Cells are linked together into larger units called lists, which are preceded by a special header cell. Lists in turn may be linked to form more complex list structures.

Special processes are required to deal with data that is linked hierarchically into lists and list structures. In addition to organizing a storage area composed of cells and assigning a unique address to each, processes are needed to deal with the storage and retrieval of information in each portion of a cell. A special list of available space should be created and maintained for storing cells that are not in use. Processes are needed for removing cells from the list of available space, and returning them when they are no longer needed.

More processes are required for linking cells together to form lists, and for maintaining information about a whole list in the header cell. Means must be provided for creating a list, adding cells to it, removing cells from it, and erasing the list when it is no longer needed. Finally, processes are required for the storage and retrieval of information on

lists, including the storage of pointers to sublists in order that complete list structures may be created and handled.

The information actually stored in cells and lists, and the ways in which cells and lists are to be linked, differs from one application to another. This is determined by establishing a representation or correspondence between the structures and processes in the intended application, and the available list structure and processes in MOSLIP. The representation established for the logic of Coursewriter II in courses is described in Chapter III.

A detailed description of the MOSLIP subprograms, along with some Fortran Coding examples, is now given.

The basic MOSLIP processes, which are described first, deal with the storage and retrieval of information in each portion of a cell. Then follows a presentation of the processes for creating and erasing cells and lists. The last subsection deals with the storage and retrieval of information on lists.

1. Basic MOSLIP structure and processes

In MOSLIP the available computer memory is organized into at most 16000 fixed-size units called cells, with each cell having a unique relative address between 1 and 16000. Cells may be linked together to form lists, each of which begins with a special header cell that contains information about the list as a whole. A list that consists only of a header is said to be empty.

Each MOSLIP cell consists of four parts which will be described individually in turn:

(a) a 2-byte integer field ID for identification

(b) a 2-byte integer field LNKL for link to the left

(c) a 2-byte integer field LNKR for link to the right

(d) a 4-byte integer field ICONT for the contents of the cell.

Figure 2.3 A MOSLIP Cell

The ID field describes the use to which a cell is put. For any cell address K between 1 and 16000, the ID field of cell K is given by the value of function IDIN(K). If this value is 2 or more, the cell is a list header. The LNKL field contains the address of the cell to the left (or "above") on the same list. For any cell address K, the LNKL field of cell K is given by the value of function LLIN(K). If K is a header cell, then LLIN(K) is the address of the cell at the bottom of the list headed by cell K, because of the circular or symmetric nature of MOSLIP lists.

The LNKR field contains the address of the cell to the right (or "below") on the same list. For any cell address K, the LNKR field of cell K is given by the value of function LRIN(K). If K is a header cell, then LRIN(K) is the address of the cell at the top of the list headed by cell K.

Subroutine SETIND is used to store values in the ID, LNKL, and/or LNKR fields of a cell. The statement CALL SETIND (I, LL, LR, K) will store the value of I in the ID field, LL in the LNKL field, and LR in the LNKR field of the cell with address K. However, if I, LL, or LR has the value -1, then the value of the corresponding field is not changed.

A simple example of the use of the MOSLIP subprograms mentioned so far follows, with notes to the right:

```
CALL SETIND (3,5,7,29)          set 3,5,7 in cell 29
CALL SETIND (-1,-1,9,29)        set 9 in LNKR of cell 29
I = IDIN (29)                   I equals the ID of cell 29
J = LLIN (29)                   J equals the LNKL of cell 29
K = LRIN (29)                   K equals the LNKR of cell 29
```

At the end of this short sequence, the value of I is 3, J is 5, and K is 9.

The ICONT field of a MOSLIP cell contains the contents or datum stored in the cell. A datum may be an integer, floating point number, Hollerith field up to 4 characters, or the name of a sublist. For any cell address K, the ICONT field of cell K is given by the value of function INHALT(K).

Subroutine STRIND is used to store a datum in the ICONT field of a cell. The statement CALL STRIND (M,K) will store the value of datum M in the ICONT field of cell with address K. For example, in the following sequence:

```
DATA M/'DATA'/, K/57/           define M and K
CALL STRIND (M,K)               store M in cell K
I = INHALT(K)                   I equals the contents of K
```

I has the value 'DATA', a 4-character Hollerith constant.

Lists may be linked together to form more complex list structure by the provision that a datum stored in a cell may be the name of a sublist. The name of a list is a simple transformation of the address of the header cell of the list. This transformation is accomplished by function NAME.

For the address K of any header cell, the name of the list headed by cell K is given by the value of the function NAME(K).

The reverse transformation from the name of a list to the address of the header cell is done by function MADLST. For the list whose name is L, the address of the header cell of list L is given by the value of the function MADLST(L). For example, in the following statement:

```
I = MADLST (NAME(K))
```

the second transformation MADLST undoes the first transformation NAME so that the resulting value of I is simply the cell address K. If L is already the address of a cell, rather than the name of a list, then the transformation MADLST(L) results in the same value L.

2. Creating and erasing cells and lists

A mainline MOSLIP program begins with two standard statements. The first is an optional declarative statement: COMMON LAVS, LW(10). This provides access to the header of a special list called the list of available space (LAVS) and to the names of ten public working lists LW(1) to LW(10). This first statement is not required if LAVS and the LW's are not referred to in the program.

The second standard statement, which isn't optional, is a call to SUBROUTINE INITAS(N), "initial available space", which organizes core into N cells and stores them on LAVS. Cells which are not in use are stored on LAVS and made available when needed. INITAS also creates the ten public working lists LW(1) to LW(10). The argument N of INITAS is equal to the number of cells created, and must be an integer between 12 and 16000. For example: CALL INITAS (16000) organizes the memory into 16000 MOSLIP cells, stores them on the list of available space, and

creates empty lists LW(1) to LW(10). The value -1 is stored in the contents of the headers of the LW's.

New cells may be obtained from the top of LAVS by function NUCELL(X), where X is a dummy variable. The address of the new cell is given by the value of function NUCELL(X). If the ID of the cell has the value 1, indicating the cell contains the name of a sublist, erasure of the sublist is called for.

Cells are returned to the bottom of LAVS by subroutine RCELL. The statement CALL RCELL(K) will return the cell with address K to LAVS. For example, consider the following statement: CALL RCELL (NUCELL(X)). In this example, a new cell is taken from the top of LAVS by NUCELL, and then it is returned to the bottom of LAVS by RCELL. Thus RCELL undoes the work of NUCELL (although the structure of LAVS is changed).

Function LISTMT(L) tests whether or not a list is empty. If L is a list name or address of a header cell, the value of function LISTMT(L) is 0 if the list is empty, and -1 otherwise.

Function MTLIST(L) empties list L by returning all its cells except the header, to LAVS. L is also the value of the function.

Function IRALST(L), "erase list", decrements the reference counter of list L by one, delivering the new value of the reference counter as the value of the function. If the decremented reference counter is 2 or less, the list L is erased by returning all its cells, including the header, to LAVS.

New lists can be created whenever desired with the LIST function. The statement L = LIST(K) creates an empty list whose name is L. The

contents of the header cell is set to the value -1. If K is 2 the reference counter of the new list is 2, otherwise the reference counter is set to 3. Only main lists or those that the programmer wishes to explicity erase when no longer needed should be created with a reference counter of 3. Sublists should ordinarily be created with a reference counter of 2, in which case they will automatically be erased when lists referencing them are erased.

Here is a short series of statements illustrating the last few MOSLIP subprograms that have been defined:

```
L = LIST(3)          create list with name L
I = LISTMT(L)        I is 0 since L is empty
J = MTLIST(L)        J equals L after emptying
K = IRALST(L)        K equals reference counter of L after erasure
```

In this example, list L is created with a reference counter of 3. Since L is empty when created, variable I receives the value 0. The third statement has no effect on list L, since L is already empty, but J receives the value of L. The last statement erases list L and sets K equal to 2.

The purpose and operation of reference counters is discussed more fully in the next section.

3. Storage and retrieval of information on lists

Two functions, NEWTOP and NEWBOT, are provided for storing data on lists. The statement K = NEWTOP(M,L) causes datum M to be "pushed down" on top of the list named L, or to the right of the cell with address L. That is, a new cell, whose relative address is K, is taken from LAVS, datum M is stored in its ICONT field, and the new cell is inserted immediately below L. If M is the name of a sublist, the reference counter of

the sublist is incremented by one and the ID field of cell K is set to 1. Otherwise, the ID field of cell K is set to 0.

Similarly K = NEWBOT(M,L) causes datum M to be "pushed up" on the bottom of list L, or above (to the left of) cell L. A new cell K is taken from LAVS, datum M is stored in it, and cell K is inserted immediately to the left of cell L, or at the bottom of list L.

Two converse functions, IPOPUP and IPOPBT, remove data that is stored on lists. Function IPOPUP(M.L) removes the cell below (to the right of) cell L, or the cell on top of list L, delivering its contents M before returning it to LAVS. The value of the function is 0 unless L is the last cell on the list (or list L is empty), in which case the value of the function is -1, no operation is performed, and M is zero.

Function IPOPBT(M,L) similarly pops up the cell above (to the left of) cell L, or the cell on the bottom of list L, delivering its contents M.

A short example is now given of the use of the storage and removal functions:

L = LIST(3)	create list name L
CALL NEWTOP(17,L)	store 17 in cell at top of list L
CALL NEWBOT(21,L)	store 21 in cell at bottom of L
I = IPOPUP(M1,L)	pop up contents M1 from top of list L
J = IPOPBT(M2,L)	pop up contents M2 from bottom of list L
K = IPOPUP(M3,L)	attempt to pop up M3 from top of list L

In this example, list L is created and the integer 17 is stored in a cell at the top, and 21 in a cell at the bottom of L. Then both the top and the bottom cells are removed ("popped"), with the result that the value of M1 is 17, M2 is 21, and I and J are both 0. List L is now empty and

an attempt to pop it up results in M3 equal to 0, and K is -1.

Sometimes it is desired to look at the datum in the top cell on a list without removing the cell that contains the datum. Function ITOP(M,L) displays datum M on top of list L, or the datum in the cell with address L. No storing or removal of cells on the list occurs. The value of the function is 0 unless list L is empty, or cell L is at the bottom of the list, in which case the value of the function is -1 and M is 0.

The address of a cell located a specified distance from the top of a list may be found using function MADNTP(L,N). The value of the function is the address of the cell that is N cells from the top of list L. For example, consider the following sequence:

```
L = LIST(3)                 create list named L
I = ITOP(M,L)               examine datum M at top of L
CALL NEWBOT(1,L)            store 1 in cell at bottom of L
CALL NEWBOT(2,L)            store 2 in cell at bottom of L
J = ITOP(N,L)               examine datum N at top of L
K = INHALT(MADNTP(L,2)      K equals datum 2 cells from top of L
```

In this example, empty list L is created and an attempt is made to examine the datum in the top cell. Since there are no cells on the list other than the header, I is -1 and M is 0. Then data values 1 and 2 are stored in cells on list L. The datum N, which receives the value 1, on top of list L is displayed, and J is 0. Then the datum K, 2 cells from the top of list L is displayed, and K is 2.

CHAPTER III

Establishing a Representation

ESTABLISHING A REPRESENTATION

The conventions by which a list structural representation of a Coursewriter course may be established are described and illustrated in Section A of this chapter. Section B summarizes the list structural codes used in the representation. A number of utility programs used in creating the representation are described in Section C. Section D contains a description of the method used to program the representation of the implicit branches generated by the input Coursewriter instructions.

A. Conventions of the Representation

In this section, conventions are described by which a correspondence is established between instructional logic and a list representation. The Coursewriter II instructions comprising course material and instructional strategy may be represented by a MOSLIP list structure according to the following four conventions.

Convention 1

Each Coursewriter II instruction is represented by one MOSLIP cell. The cell contents are the operation code and modifier of the Coursewriter II instruction, a total of three characters. Labels, which may be considered to be minor instructions, are up to 12 characters long, and hence are represented by as many as 3 consecutive cells on a MOSLIP list.

In general, it is not necessary to represent the details of an instruction as specified by its parameters and subparameters, since these usually have no effect on the instructional strategy or path

which a student would follow through the course. However, this generalization is not true of the br instruction, nor the instructions ad, sb, mp, dv, lr, and ld which change the values of counters, switches, and return registers, and hence affect branching. The parameters of these instructions are stored in subsequent MOSLIP cells on the same list, but each parameter is preceded by a blank character to provide one column of indentation on printout.

Occasionally, on account of the implicit branching of the course flow decision table, two or more paths with different conditions may develop in the same sequence of Coursewriter instructions. In such a case, each path receives a separate list representation, and all of the instructions in the sequence appear on each path representation.

Convention 2

Sequential execution of instructions along any path through a course is represented by consecutive cells on a list, from left to right (top to bottom).

Convention 3

Each label and each control instruction (major instruction or ep) begins a new list. This means that the MOSLIP cell representing the label or instruction is preceded by a list header cell. Execution of such an instruction is indicated by a cell pointing to the list header. This may imply a merger of one or more paths. The reason for this provision is that such instructions can, in general, be the destination of explicit or implicit branches.

A minor instruction following a synonymous answer instruction (wb, cb, or ab) also starts a new list since it may be branched to as a result of a "no execute, examine next instruction" condition.

Examples 3 and 4 below illustrate this situation.

If the same label or minor instruction occurs on more than one path generated by the course flow decision table, a header cell occurs on only one such path.

Convention 4

A branch is a departure from sequential execution and is represented by a pointer to a sublist. Such branches may be either conditional or unconditional and either explicit or implicit. In the case of conditional implicit branches due to mismatch or time-out on a response analysis or ep instruction respectively, the pointer cell always immediately follows the cell containing the instruction.

Examples illustrating the application of the four conventions are now given. Figure 2.2 of Chapter II also illustrated some of the conventions of the representation. In the examples, the Coursewriter II sequence of instructions is given first, followed by its MOSLIP representation. Each MOSLIP cell is indicated by a box and its contents are printed in the box. P indicates a cell pointing to a sublist headed by a header cell designated H.

Example 1 - Unconditional explicit branch

In this example, the label label2 has 6 characters, so its representation requires two consecutive MOSLIP cells, with the first 4 characters in the first cell, and the last 2 in the second cell. The label is preceded by a header so that it may be branched to from other points in the program:

```
dt              display text
br label2       branch to label2
```

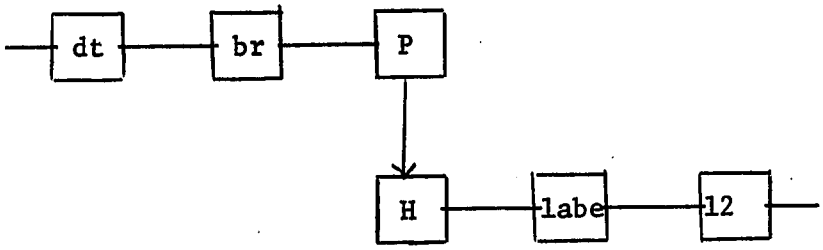

Figure 3.1 Logic of Example 1

Example 2 - Conditional explicit branch.

The parameters of the conditional br are each indented or preceded by a blank. The cell pointing to the label immediately follows the parameters of the condition:

br label ¬/c1 ¬/1e ¬/3	branch to label if counter 1 is less than or equal to 3
dt	display text

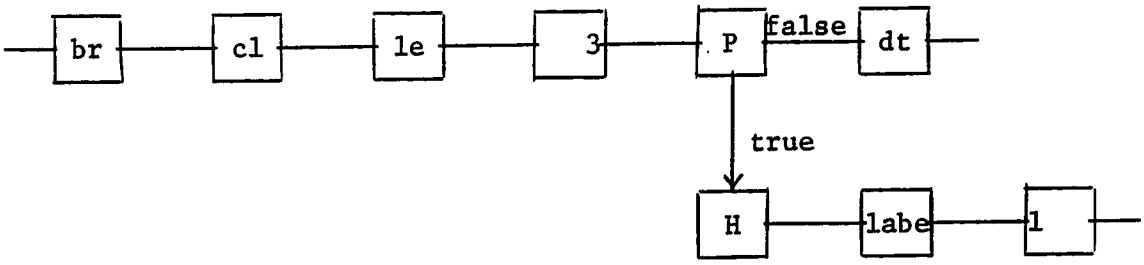

Figure 3.2 Logic of Example 2

Example 3 - Implicit branches

This example shows both conditional and unconditional implicit branches. Following the fourth convention, the ca mismatch pointer cell immediately follows the cell containing the ca, whereas the match condition involves only sequential execution and hence is indicated by subsequent cells on the same list. This list ends with an unconditional implicit branch to the minor pa, on account of the "no execute examine

next instruction" condition that occurs when the cb is encountered on this path. The cb instruction is executed in case of ca mismatch. If there is a match on the cb, the pa is executed first, otherwise execution skips immediately to the pr. The list structure clearly shows three paths through this sequence:

ca	correct answer
dt	display text
cb	synonymous correct answer
pa	pause
pr	start new problem

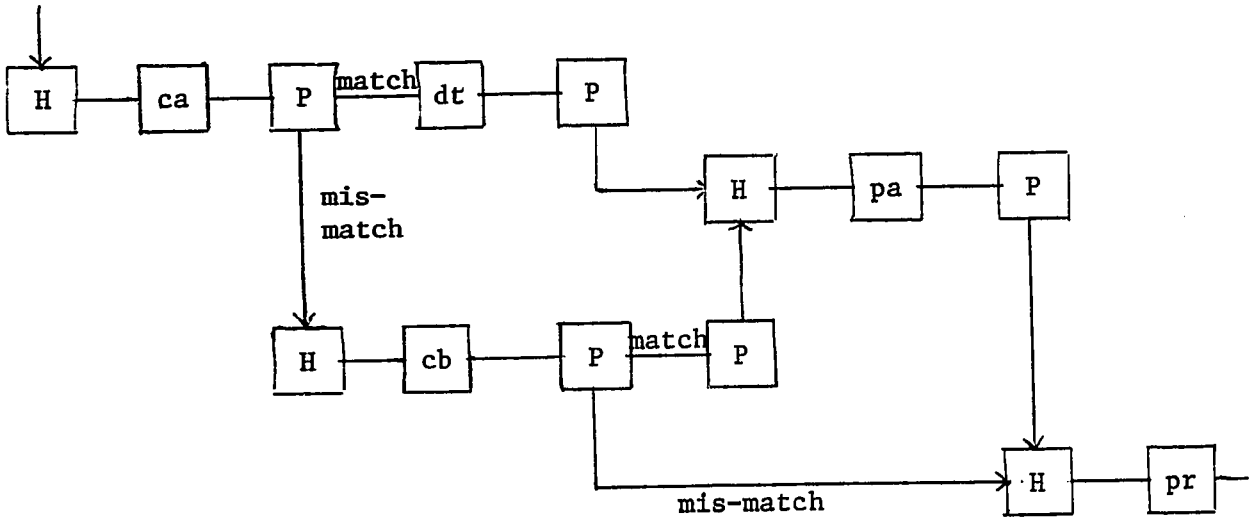

Figure 3.3 Logic of Example 3

Example 4 - Two paths through the same sequence

In the discussion of the first convention it was mentioned that two or more paths could develop through the same sequence of instructions. An example of this situation is the pa instruction in the following sequence. The pa instruction has two separate representations because it appears on two paths under different conditions with different logical

outcomes. When pa is executed following a match on ab, the nx is encountered but not executed as a skip occurs to the next major (pr). However, when pa is executed following execution of the first nx, ab is skipped ("no execute, examine next instruction") and the second nx is executed. Thus the pa occurs in two cells, only one of which is preceded by a header.

aa	additional answer
dt	display text
nx	conditional execute
dg	display graphic
ab	synonymous additional answer
pa	pause
nx	conditional execute
dl	display emphasis line
pr	start new problem

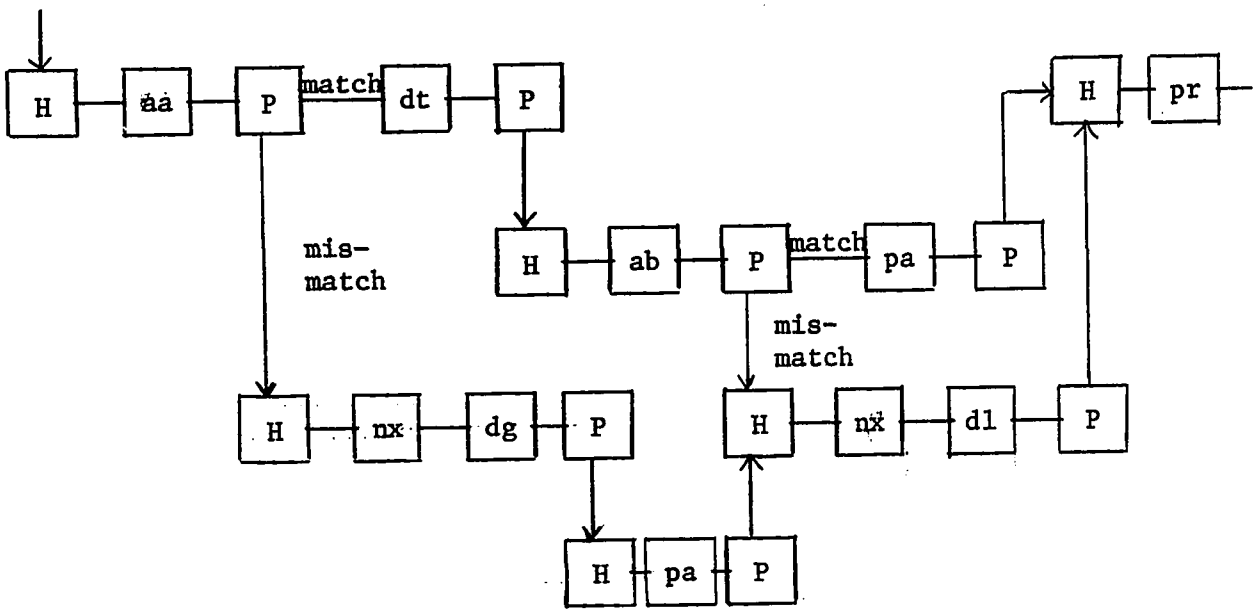

Figure 3.4 Logic of Example 4

B. List Structural Codes

In order to set up a MOSLIP representation of an arbitrary input Coursewriter II course program, the four conventions described in the previous section need to be applied. In addition to meeting the conventions, the representation should store all necessary information in order that the instructional logic of the course can be effectively traced and simulated. To meet these requirements, codes are established for the Coursewriter II instructions affecting branching, and for all the conditions occuring in the course flow decision table (Figure 2.1). In addition, a small number of special codes are created to represent certain list structural situations and information for printing a logic chart. All codes used are integers, positive or negative.

The operation code for Coursewriter II instructions affecting branching is given in Figure 3.5. There are 20 such instructions, including all majors and ep, br, tr, ad, sb, mp, dv, lr, ld (except load buffer), and fn. Each such instruction is coded as an integer between 1 and 20.

ea	nx	aa	ca	wa	un	pr	wb	cb	ab
1	2	3	4	5	6	7	8	9	10

ep	br	tr	ad	sb	mp	dv	lr	ld	fn
11	12	13	14	15	16	17	18	19	20

Figure 3.5 Coursewriter Instruction Code

The Coursewriter instruction code is used in a number of ways to establish the list structural representation of a sequence of Coursewriter

instructions. Initially, the code is used by the mainline program to identify an instruction. The instruction code is then stored in the list structure, along with the instruction itself.

A control instruction (major or ep) is, according to the third convention, preceded by a list header. This is true also of every label, and of each minor instruction following a synonymous answer (wb, cb, ab). The instruction code is stored in the contents (ICONT field) of the header as indicated in Figure 3.6.

Code	Meaning
-12 to -1	Label (negative character length)
-1	single character label or any other list
0	minor other than ep
1	ea
2	nx
3	aa
4	ca
5	wa
6	un
8	wb
9	cb
10	ab
11	ep
more than 100	pr (in sequence)

Figure 3.6 Header Contents Code

The length of a label, up to 12 characters, is stored as a negative integer in the contents of the corresponding header cell. The code -1 may refer to a single character label, or it may be the contents of any other list, such as the working lists LW(1) to LW(10). The header preceding a control instruction contains the corresponding instruction code, 1 to 11,

except for the pr instruction. Problems (pr) are given a sequence number in the order in which they are encountered in a course, beginning at 101, and this sequence number is stored in the contents of the header of each pr list.

The remaining Coursewriter II instructions that affect branching, coded 12 to 20 in Figure 3.5, are not preceded by a list header. Instead the instruction code is stored in the ID field of the MOSLIP cell that contains the instruction. The ID or identification field is used for a number of purposes as summarized in Figure 3.7.

As indicated in Figure 3.7, an ID of 2 or more identifies a cell as a list header. The amount by which the ID exceeds 2 is the value of the reference counter, which is the number of times the list is referenced as a sublist. This represents the number of possible paths of execution along which subsequent instructions lie. The reason for the header is given in the ICONT field of the same cell, coded as in Figure 3.6.

An ID of 1 denotes that the cell contains the name of a sublist representing an instruction encountered through merging or ordinary sequential execution, according to the third convention. An ID of 0 indicates that the cell contains 1 word of Hollerith characters.

Departures from sequential execution are represented by pointers to sublists according to the fourth convention. The type of branch is coded in the ID field of the pointer cell containing the name of the sublist. An ID of -3 or -4 identifies a cell containing the name of a sublist to which a branch occurs on condition of time-out or mismatch respectively. Any other departure from sequential execution is designated by a pointer cell with an ID of -2.

Code	Meaning
2 or more	header and reference counter
1	merge to name contained
0	contains 1 word characters
-2	branch to name contained
-3	conditional branch: ep time-out
-4	conditional branch: mismatch
-5	contains an integer
-6	contents and subsequent cell form 2 word characters
-7	contents and subsequent 2 cells form 3 word characters
-8	suppress printing of contents
-12	conditional br
-14	ad
-15	sb
-16	mp
-17	dv
-18	lr
-19	ld

Figure 3.7 Cell ID Code

An ID of -5 indicates that the cell contains an integer, instead of characters. This is used, for example, to store parameters of instructions such as ad or ld that refer to the values of counters or switches.

As already mentioned, a cell containing characters in its ICONT field has an ID of 0, unless the characters form a unit whose length exceeds 4. In the latter case, the unit requires more than one cell for its storage. As many as three consecutive cells are used to store labels, which may be as long as 12 characters. The ID of the first cell indicates the number of cells required, with -6 denoting 2 consecutive cells, and -7 denoting 3 cells. The ID of each of the extra cells is set to 0.

Two cells are also required to store a reference to a switch such as S10a, which is 4 characters long, but is preceded by a single blank character for indentation during printing, making a total of 5 characters. In this case, the blank and the first three characters are stored in a cell with an ID of -6, and the last character is stored in the next cell on the same list, with an ID of 0.

Occasionally it is desirable, during printing of the logic chart, to suppress the printing of the contents of a cell. The ID field of such a cell is set to -8. For example, an ID of -8 is given to the cell containing the value of the IUNI counter, which is a count of the number of un instructions for each un following an ep. This is an internal counter used only for determining the un (counter) match or mismatch condition. Comparison is made with the IUN2 counter whose value is equal to the number of times the ep has been executed. An ID of -8 is also used for the cell that stores the address of a label loaded into a return register by the lr instruction.

An ID of -12 indicates that a cell contains a conditional br instruction. The parameters or items forming the condition follow in subsequent cells, each indented by one blank character.

ID values of -14 to -19 indicate that the contents of a cell is an instruction coded 14 to 19 respectively, according to the instruction code (Figure 3.5). These are the instructions ad, sb, mp, dv, lr and ld.

Examples are now given illustrating the use of the header contents code (Figure 3.6) and the cell ID code (Figure 3.7). In the following four diagrams (Figures 3.8 to 3.11), a cell is shown as a box divided into two parts, with the upper part containing the ID or identification field,

and the lower part the ICONT or contents field. Characters are shown left justified, while integers are right justified (as far to the right as possible) within their part of a box. Pointers to sublists are again designated P, but headers are identified merely by an ID greater than or equal to 2.

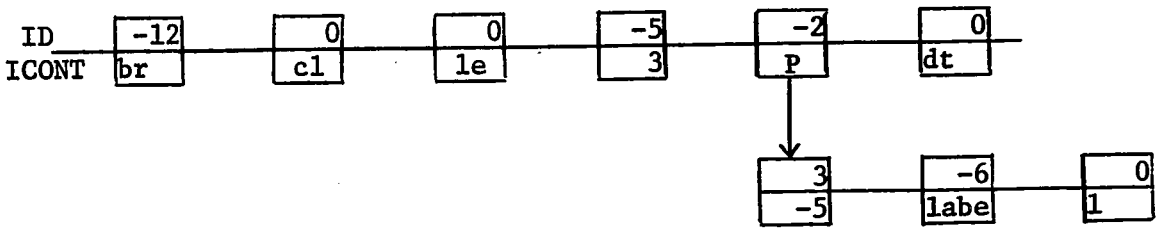

Figure 3.8 Coding Example 2

Figure 3.8 is an elaboration of example 2 of the previous section, showing the complete coding of the ID and ICONT fields of the MOSLIP representation of the sample of Coursewriter coding. The first cell has an ID of -12 and contains a conditional br instruction. The following two items, c1 and 1e, are each indented by one blank character and have an ID of 0. The last item of the condition is the integer 3 contained in a cell with an ID of -5.

In case the condition is true, the pointer cell follows with an ID of -2 indicating a departure from sequential execution. Otherwise, dt is executed, as shown stored in a cell with 0 ID. When the condition is true, the branch is to label, headed by a header cell with an ID of 3, indicating no other references to this label. The contents of the header cell is -5, indicating a label of length 5 characters. The label label is contained in two consecutive cells, the first of which has an ID of -6, indicating that another cell follows, with an ID of 0.

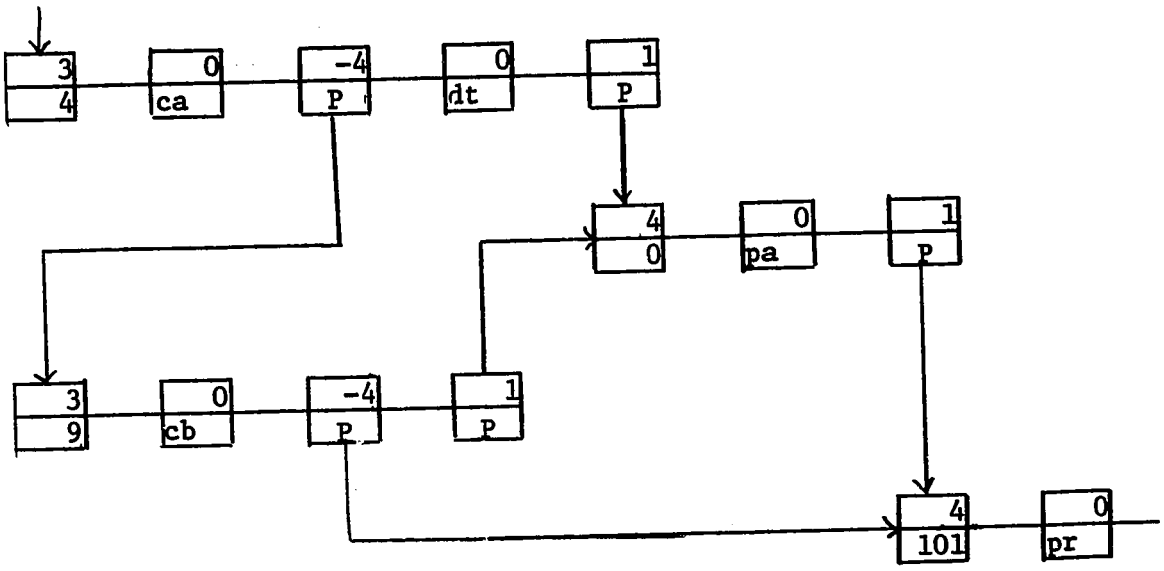

Figure 3.9 Coding Example 3

Figure 3.9 is an elaboration of Example 3 of the previous section. The ca instruction is preceded by a header cell with a contents of 4, which is the ca instruction code (Figure 3.5). In case of mismatch, a pointer cell with an ID of -4 indicates a branch to cb, headed by a header cell with an ID of 3 and contents of 9. In case of a match on the ca, the dt is executed and then a merge takes place to the pa instruction as indicated by a pointer cell with an ID of 1. The merge is with the path generated by a cb match condition. The reference counter of the header cell preceding the pa instruction has been incremented twice and hence the ID is 4, while the ICONT field is 0 indicating a minor instruction. The pa is followed by sequential execution of pr, as indicated by a pointer with an ID of 1. The header of the pr instruction also has an ID of 4, since merger takes place with the path generated in case of mismatch on cb. The pr header has a contents of 101, indicating the first pr encountered in the course.

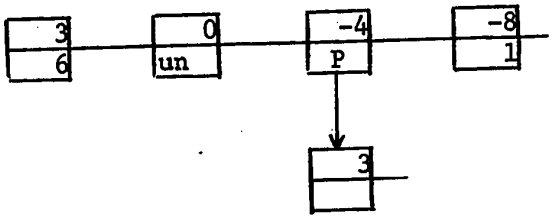

Figure 3.10 Coding an un Instruction

Figure 3.10 shows the coding for the first of a series of un (unrecognizable response) instructions. The un is preceded by a header cell containing a 6 in the ICONT field in accordance with the header contents code of Figure 3.6. The value of the IUNI counter is 1, indicating the first of a series of un instructions, and the integer 1 is shown in a cell with ID equal to -8 to suppress printing. When the un is executed, counter IUNI is compared with internal counter IUN2, whose value is equal to the number of times the preceding ep instruction has been executed. If the counters match, the instructions following the un are executed, otherwise a branch is made as shown by the pointer P.

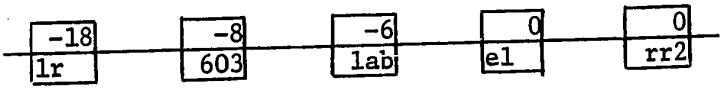

Figure 3.11 Coding an lr Instruction

Coding of the instruction lr label ¬/rr2 is shown in Figure 3.11. The lr instruction is contained in a cell with an ID of -18, in agreement with the cell ID code of Figure 3.7. The label address 603 is stored next in a cell with ID of -8 to suppress printing. Label addresses are generated by function NULABL discussed in the next section. The label is stored in the next two cells, the first of which has ID equal to -6 to indicate that another cell follows. The label is preceded by a blank

character in the first of the two cells to provide for indentation during printout. The item rr2 is also preceded on the left by a blank character in the last cell.

A complete summary of the utilization of the four fields of a MOSLIP cell for simulating or representing the logic of a Coursewriter II course is given in Figure 3.12. In this figure, the four fields of a cell are shown as columns of the table, and the possible values in each field are in the rows of the table. The ID or identification field provides summary information about the contents (ICONT field) of the cell, while LNKL and LNKR always store the addresses of the cells linked to the left and right respectively. Figure 3.12 combines and relates Figure 3.6 and Figure 3.7.

C. Utility Routines

The list structural representation of an arbitrary input sequence of Coursewriter II instructions is created in one pass. This is done by a mainline program, CL2, and function MAJOR discussed in the next section on implicit branches. In order to create a list structure in agreement with the conventions and codes just described, a number of more general processes, referred to here as utility routines, are required.

Character manipulation operations required for establishing the representation include character packing and unpacking. Packing is necessary in order to store condensed information in cells, but occasionally unpacking must be done to place data in suitable form for printing. Whole character strings may need to be packed, and the length of strings up to specified delimiters determined. Integers need to be computed from their character representation in order that arithmetic may be performed. The reverse operation of computing the character representation corresponding

ID	LNKL	LNKR	ICONT
2 or more header and reference counter	1 to 16000 link left	1 to 16000 link right	more than 100 -- pr (in sequence order) 11 -- ep 10 -- ab 9 -- cb 8 -- wb 6 -- un 5 -- wa 4 -- ca 3 -- aa 2 -- nx 1 -- ea 0 -- minor other than ep -1 -- single character label or any other list -1 to -12 -- label (negative character length)
1			merge to name contained
0			contains 1 word characters
-2			branch to name contained
-3			conditional branch: ep time out
-4			conditional branch: mismatch
-5			contains an integer
-6			contents and subsequent cell form 2 word characters
-7			contents and subsequent 2 cells form 3 word characters
-8			suppress printing of contents
-12			conditional br
-14			ad
-15			sb
-16			mp
-17			dv
-18			lr
-19			ld

Figure 3.12 MOSLIP Cells for Coursewriter II Representation

to an integer is also required in order that all information can be printed in a common format.

Two general list processes required, in addition to those provided by MOSLIP, include searching and copying. When creating the list structure and printing it, lists are set up that store information about the operations, and these must be searched at later times to find the data that has been stored. Copying is necessary when more than one path develops through a sequence of instructions.

Finally, processes are required to dissect and store labels and instructions in accordance with the conventions and codes. Labels must be identified, stored, and assigned a unique address. For instructions that affect branching, the parameters must be separated, interpreted and stored.

Programs that carry out these utility processes are now described individually in more detail.

1. Character Manipulation

Character manipulation operations are performed by five subprograms: PACK, UNPACK, SPACK, LIMTER, and HOLLER. Four characters may be packed into a 360 word by the function PACK(I1, I2, I3, I4). The four arguments I1 to I4 are the four characters to be packed, each left justified in its own word. The value of PACK is the resultant packed word with a floating-point or real name. Entry IPACK(I1, I2, I3, I4) may be used if the packed word is to have a fixed-point or integer name.

The inverse function is UNPACK(I, IB). The symbol I represents the word to be unpacked, and IB is the byte number, 1 to 4 only. The value of UNPACK is the character that is unpacked from byte IB of word I, left

justified with blanks in the rightmost three bytes. UNPACK is used if the unpacked character is to have a real name, otherwise entry IUNPAK(I,IB) unpacks byte IB from word I and gives the result an integer name. Both PACK and UNPACK require subroutine EQUAL(Y,X) which sets Y equal to X regardless of real and integer naming conventions. Arguments I1, I2, I3, and I4 of PACK, and argument I of UNPACK, can be either real or integer.

For example, the value of PACK(IUNPAK(X,1), IUNPAK(X,2), IUNPAK(X,3), IUNPAK(X,4)) is X. As another example, the value of IPACK(UNPACK('SKI',1), UNPACK('SUN',3), UNPACK('POLE',2), UNPACK('TOW',3)) is 'SNOW'. The effect of PACK is to change 4 characters from A1 format to A4 format, while UNPACK changes 1 character from any position in A4 format to A1 format.

A vector string of characters in A1 format may be packed into a vector of words of A4 format using subroutine SPACK(STRING, LENGTH, PACKED, N). Input is a character vector STRING of a specified LENGTH in A1 format, with each character left-justified in its word. Output is a PACKED character vector string of length N words, with 4 characters per word in A4 format. The last or N'th word of PACKED contains up to 3 blanks to the right in order to fill up the word. For example, if STRING(1) to STRING(6) have the values 'S','T','R','I','N','G' respectively, then the statement CALL SPACK (STRING,6, PACKED,N) produces PACKED(1) equal to 'STRI', PACKED(2) equal to 'NG', and N equal to 2.

The length of a character vector string of format A1, up to a specified delimiter, is found by function LIMTER(STRING, LENGTH, DELIMT,M,N). The input character vector STRING has a maximum total LENGTH in A1 format with each character left-justified in its word. The delimiter DELIMT consists of M consecutive characters. The maximum M is 4 characters left-justified

and packed into the word DELIMT. The value of the function is also the value of N, which is the length of the string up to but not including the delimiter. For example, if STRING(1) to STRING(8) have the values 'S', 'T','R','I','N','G','¬','/' respectively, STRING(9) to STRING(80) are any characters, and the delimiter is '¬/' of length 2, then the value of LIMTER(STRING,80'¬/',2,N) is 6, and N also receives the value 6.

Some common delimiters occurring on Coursewriter II cards are the 0-8-2 punch (enter symbol) denoting the end of a statement, '¬*' denoting continuation of a statement on a subsequent card, and '¬/' which separates the parameters of an instruction.

An integer of magnitude less than 100,000 receives a character representation for each of its digits by subroutine HOLLER(IHOLL,INT,IP). The input integer INT is converted to a packed character string IHOLL of length 2 words of type A4, and to an unpacked character string IP of length 8 words of format A1. The individual digits of the integer are left-justified in each string, with blanks filling any remaining space to the right. For example, if the integer INT has the value 769, then the statement CALL HOLLER(IHOLL,INT,IP) produces IHOLL(1) = '769', IHOLL(2) is blank, IP(1) is '7', IP(2) is '6', IP(3) is '9', and IP(4) to IP(8) are blank.

2. List-processing subprograms

Two list-processing subprograms of general applicability are IFIND and LCOPY. Function IFIND(M,L,K) searches list L from the top for a cell containing datum M. If such a cell is found, then K is the cell address and the value of the function is 0. If the datum M is not found on list L, then the value of the function is -1.

Function LCOPY(ILST,NLST) copies the ID and contents of consecutive cells on a list beginning with the first cell below ILST, and storing the copied cells on the bottom of the list with address NLST. The value of the function is NLST. Pointer cells with an ID of 1 are not copied, but descent is made into the indicated sublist and its cells are copied onto the bottom of NLST. This subprogram is used by function MAJOR to copy a sequence of instructions when two or more paths with different outcomes develop by implicit branching.

3. Dissecting and storing instructions

Labels are identified, stored, and assigned a unique address by function NULABL(LABEL,N). The character vector LABEL of length N is packed from A1 format into temporary storage in A4 format. Working list LW(7), which stores the addresses of all labels, is searched and if LABEL is found its address is returned as the value of the function. If LABEL is not found on LW(7), then a new list is created to represent the label, LABEL is stored in packed form on the new list, and the address of the new list is stored on LW(7) and returned as the value of the function. NULABL also stores the appropriate codes in the label header and in the cells on the new list that is created if necessary. Function NULABL is called by subprograms BR and LR as well as by the main program.

The main program, given the name CL2, reads Coursewriter II program cards one at a time into the integer vector COL in format (80A1). The label or instruction is identified and, if it is to be executed, it is stored in a new cell on the bottom of the current execution list LST. Then subroutine LR, BR, TR or DISECT is called in case the instruction is lr, br, tr or one of (ad,sb,mp,dv,ld) respectively. These subroutines

dissect and store the parameters of the corresponding instructions, according to the conventions and codes previously described. The parameters are stored in consecutive cells on the current list.

Subroutine BR sets up an explicit branch to a label, last executed ep, a pr, or a return register. First the subroutine determines whether the branch is conditional or unconditional. For a conditional br, a flag is reset and subroutine BRC is called to store the items making up the condition. Then subroutine BR determines the destination of the branch, which is the address of the header of a sublist according to Convention 4.

For an explicit branch to a label, subroutine BR calls NULABL. If the destination of a branch is the last executed ep, working list LW(9) is designated as the sublist. LW(9) holds the address of the last encountered ep, which is updated by subroutine MAJOR. If MAJOR cannot determine from the static logic that the last executed ep is the same as the last encountered ep, then LW(9) is set equal to the "TO EP" list LW(10), as described in the next section.

For an explicit branch to a pr, working list LW(8) is searched to find the appropriate sublist. LW(8) holds the addresses of the current pr and the next seven consecutive pr's that may be branched to. Subroutine MAJOR updates LW(8) whenever a pr is encountered in the input data. If the branch is to a return register, subroutine BR designates sublist LW(1) to LW(6), representing return registers rr0 to rr5 respectively. Working lists LW(1) to LW(6) contain only one cell, which holds the name of the correponding return register.

Subroutine TR sets up a transfer to a label in another segment. A new list is created to represent each such label encountered, and these labels

Last executed control instruction and/or current condition

Currently Encountered Instruction

Condition Code →		1	2	3	4	5	6	7	8	9	10	11
Instruction Code	Instruction	un (counter) mis-m	ep timed out	aa,ab mis-m	ca,cb mis-m	wa,wb mis-m	pr nx ea	aa,ab match ep in time	ca,cb match	wa,wb un (counter) match	skip to next major	after explicit branch to label
1	ea	1	2	2	2	2	2	2	2	5	0	0
2	nx	0	0	0	0	0	0	4	2	5	0	0
3	aa	0	0	0	0	0	0	0	2	5	0	0
4	ca	0	0	0	0	0	0	0	2	5	0	0
5	wa	0	0	0	0	0	0	0	2	5	0	0
6	un	0	0	0	0	0	0	0	2	5	0	0
7	pr	1	0	0	0	0	0	0	0	5	0	0
8	wb	3	3	3	3	0	3	3	3	3	0	0
9	cb	3	3	3	0	3	3	3	3	3	0	0
10	ab	3	3	0	3	3	3	3	3	3	0	0
11	ep minors	3	3	3	3	3	0	0	0	0	3	0

Figure 3.13 Decision Table Code

(see Figure 3.14 for explanation of numbers in the table)

Code	Execution Decision
0	execute
1	no execute -- return to un
2	no execute -- skip to pr
3	no execute -- examine next instruction
4	no execute -- skip to next major
5	no execute -- return to last executed ep

Figure 3.14 Execution Decision Code

are not stored on the labels working list LW(7). Subroutine TR stores on the new list both the name of the label and the segment number to which the transfer is indicated.

D. Implicit Branching

Most of the work of determining implicit branches and setting up their list representation is done by subroutine MAJOR. The nature and location of implicit branches is determined from the Course Flow Decision Table (Figure 2.1), which is stored as a table of coded integers (Figure 3.13) in subroutine Major.

In the Decision Table Code of Figure 3.13, the currently encountered instruction is found along the rows, which are numbered 1 to 11 as in the Coursewriter Instruction Code of Figure 3.5. The last executed instruction and/or current condition is found in the columns of Figure 3.13, while the decision to be taken with respect to the current instruction is found in the body of the table. The decisions themselves are coded as integers 0 to 5, with meaning as explained in Figure 3.14, the Execution Decision Code.

In the Execution Decision Code of Figure 3.14, the decision to execute is coded 0, while no execute is coded 1 to 5 depending on the destination of the implicit branch, if any. Codes 1, 2, 4 and 5 designate the execution of an instruction other than the current one, while code 3 indicates that the next instruction is to be examined with no change in the current condition. Decision code 4 - skip to next major - is also included in the Decision Table Code, column 10, while column 11 indicates that any instruction is executed following an explicit branch to a label.

The implication of the Decision Table Code of Figure 3.13 is that one or more paths of execution and/or non-execution may develop automatically or implicitly in a given sequence of Coursewriter instructions. This may be seen particularly from row 11, which applies to all minor instructions as well as ep. Row 11 consists only of 0's and 3's, indicating that either a minor instruction is executed, or else the next instruction is examined. Along any path, a consecutive series of minor instructions is either executed, or else merely examined until the next major instruction is encountered. This means that all paths may be classified into two types: paths along which a minor instruction will be executed ("continuation" paths), and paths along which a minor instruction will not be executed but the next instruction will be examined ("branch" paths).

Continuation paths (columns 6 to 9 and column 11 of Figure 3.13) and branch paths (columns 1 to 5 and 10) develop from the execution of a major instruction or ep, or by explicit branch to a label (column 11 only). The branch and continuation conditions so generated form the columns of the Decision Table Code of Figure 3.14. A complete list of these generated conditions and their origin is given in Figure 3.15.

Code	Instruction	Branch Condition 1	Continuation Condition 2
1	ea	0	6
2	nx	10	6
3	aa	3	7
4	ca	4	8
5	wa	5	9
6	un	1	9
7	pr	0	6
8	wb	5	9
9	cb	4	8
10	ab	3	7
11	ep	2	7
	label		11

Figure 3.15 Branch and Continuation Conditions

Figure 3.15 lists the conditions of the branch and continuation paths generated by each control instruction and label. Execution of any one of these instructions produces a continuation condition (condition 2) whose code is given in the last column of Figure 3.15, and which is used to determine the appropriate column of the Decision Table Code of Figure 3.13 when a new instruction is encountered. Most of the instructions of Figure 3.15 also generate a branch path whose condition is given in the previous column (condition 1). However, ea, pr, and labels do not generate branch paths, so their branch condition is coded as 0. The branch condition code is also used to determine the appropriate column of Figure 3.13 to find out the execution status of subsequent instructions that are encountered.

In order to keep track of the multiplicity of paths and conditions which could potentially develop through implicit branching, two current lists LCURR(1) and LCURR(2) are created and used to store essential information about branch and continuation paths respectively. LCURR(1) and

LCURR(2) store consecutively the list header address of the last control instruction or label executed, and the corresponding branch or continuation code. Any number of path headers and their conditions can be stacked on each list, with the most recent on top.

The mainline program CL2 creates lists LCURR(1) and LCURR(2), and initializes LCURR(1) to the list representing the beginning of the sequence of Coursewriter instructions whose representation is to be established. CL2 also stacks the label header address and condition code 11 on top of LCURR(2) whenever a label is encountered, assuming that the label may be branched to from elsewhere in the course. For the same reason the last executed ep list LW(9) is set equal to the more general "TO EP" list LW(10) after a label is encountered, since the label may be branched to after execution of some ep other than the last ep encountered. Working list LW(10) contains only two cells storing the information "TO EP", and no attempt is made to determine which ep is to be represented. The appropriate ep can be determined when the list structural representation is later traced. Whenever a minor instruction is encountered, it is stored only on the topmost continuation list of LCURR(2).

Subroutine MAJOR is called whenever a control instruction, TR, or unconditional BR is encountered. Current lists LCURR(1) and LCURR(2) are popped to determine the outcome of each current branch or continuation path that has not yet been disposed of. Each continuation list is checked to see whether or not its execution outcome is the same as the previous continuation list above. If the outcome or decision, as determined from the decision table code, is the same, then, unless the decision is to examine the next instruction, the first or upper list is stored on the second or

lower list. Thus a merger of paths is created in agreement with Convention 3. If the decision with respect to two continuation paths stacked consecutively on LCURR(2) is different, or if the decision is to examine the next instruction, then the upper continuation list is copied onto the lower by function LCOPY described in the previous section.

Function MAJOR determines a list LISTEX to represent the currently encountered control instruction, if it is to be executed. All branch or continuation paths are then disposed of in terms of their outcomes from the Decision Table Code of Figure 3.13. MAJOR keeps track of the last encountered ep, LW(9), and the last un, LASTUN, as well as updating the pr storage list LW(8) whenever a pr is encountered. If the decision for a branch path is to examine the next instruction (code 3 of the Execution Decision Code of Figure 3.14), then the path list header and its condition are stored back again on the same current list LCURR(1), in order that they may be dealt with later. If the decision for a continuation path is to examine the next instruction, then a new list is created for the first minor that may follow, and it is this list and the unchanged condition which are stored back on the current list LCURR(2). This provides for the latter part of Convention 3.

Finally, function MAJOR stores on each branch or continuation list the outcome or decision list which has been determined for the path under the existing conditions. The new branch and continuation conditions which may have been created by execution of the currently encountered instruction are stored on the corresponding current lists LCURR(1) and LCURR(2), which are thus continually updated.

Function MAJOR is called in case of a TR or unconditional BR simply

to dispose of all current continuation lists on LCURR(2). The continuation lists will in this case all become sublists of each other as they are unstacked from LCURR(2), and LCURR(2) is left empty.

CHAPTER IV

Applications of the Representation

APPLICATIONS OF THE REPRESENTATION

This chapter presents a description of three applications of the list structural representation of a Coursewriter II sequence established in Chapter III. The first application, discussed in section A, is the development of a printed logic chart illustrating the instructional logic of a course. Tracing all possible paths that could be followed by a student through a course is discussed in section B. The third application, described in section C, is the representation of student response models and the determination of simulation variables for each artificial student.

A. Logic Chart

The purpose of printing a logic chart is to assist authors with the analysis of the instructional logic of a course, as well as to be an aid in debugging and documentation. Thus the objective of the logic chart is to provide a clear representation of course logic, while omitting unnecessary detail. A diagrammatic two-dimensional logic chart was decided upon in order to provide the most visually useful output, as well as to exploit the full potential of the paper. This is in contrast with most types of computer-drawn flowcharts, which are one-dimensional (Sherman, 1966).

1. Chart Specifications

The specifications or conventions according to which a logic chart may be developed are now described:

(a) All Coursewriter instructions associated with a particular pr are printed separately. The reason for this is that the pr

instruction is designed to specify the beginning of a new problem, and thus the pr and associated instructions are intended to form a logical unit. Each problem unit is clearly separated.

(b) A series of sequentially executed instructions are arranged vertically in one of a number of channels or columns down the printed page, The vertical representation for sequential execution is natural in that it corresponds to the way the instructions are originally composed. Several vertical channels are provided in order that branches and merges among channels may be diagrammed.

(c) In general, only the operation code of an instruction is printed, but any control instruction or minor which is represented by a list header, has the address of the header printed above the instruction, for reference purposes. The same reference number is printed in the left margin of an augmented course listing which is output before the logic chart. The reference number (header address), where present, is preceded above by a vertical bar (|), in order that the location of the instruction may stand out for purposes of cross reference. Conditional explicit branches and any instructions that change the values of counters, switches, or return registers are followed by the instruction parameters or items, each indented one column. Printing the parameters is desirable since the detailed information affects branching, while the indentation serves to separate instructions from parameters of instructions.

(d) Labels are also preceded above by a vertical bar to make them stand out for cross reference. Occasionally the logic of implicit branching requires the execution of a sequence of instructions under two

or more different conditions. Each such path is indicated separately in the chart, and the instructions and their associated labels appear on each path. Thus a series of instructions and labels may appear at more than one place on the chart. A label is preceded by a vertical bar on only one such path, indicating the point at which branches or merges are to take place. Also, any control instruction or minor which possesses a reference number (address of list header) is preceded above by its reference number and a vertical bar on only one path. This convention avoids any possibility of ambiguity.

(e) Departures from sequential execution (branches) are to be indicated by strokes (---) to another vertical channel. A conditional branch due to time-out on a response request (ep) is indicated by the printing of T-OUT along the strokes, while a branch due to mismatch during response analysis is indicated by MIS-M. An explicit branch due to an unconditional br is indicated by strokes beside the br. For a conditional br, the items forming the condition are indented and printed below the br, while the destination of the branch is indicated by strokes beside the last item of the condition. Any other type of branch, also indicated by strokes, results from the implicit branching feature of Coursewriter II.

Sometimes when a branch is called for, a channel may not be able to indicate its branch until adjacent channels have cleared. This waiting is indicated by continuing the channel with a series of vertical bars until the branch can be displayed.

(f) Merger of paths. Paths through a course do not ordinarily terminate except by transfer to another course segment. Instead, when a vertical channel of instructions on the logic chart comes to an end,

then, if no branch is indicated, a merger takes place with another path. The point of merger is indicated by the label or numbered statement which terminates the channel.

A merger, in contrast to a branch, does not constitute a departure from sequential execution. Instructions following a label, control instruction, or minor with a reference number (list header) are printed only once. If the instructions can be reached or executed from some other point in the course, this is indicated by branch or merge, but the subsequent instructions are not printed again.

(g) When a branch occurs back to the last executed ep, the program may not have determined which ep was executed last for each path. In this case, the branch is indicated as "TO EP", without specifying which ep. Similarly, branches to return registers are indicated as such, regardless of which label may have been loaded last into the return register.

(h) Because some labels may not be branched to, or may be branched to only through return registers, any labels and associated instructions not previously printed will be printed when the rest of the logic chart has been completed.

(i) Function call instructions (fn) with modifiers are represented in accordance with the appropriate row of the Course Flow Decision Table (Figure 2.1) as indicated by the modifier. Row 2 of this table is ambiguous (aa, ca, wa, or un), so that a user-supplied routine FN2 is provided in the program to specify the behavior intended when fn is used with modifier 2.

(j) The call macro (cm) instruction is treated as a minor having no effect on branching. Therefore macros should be in expanded form in the input deck.

2. Programming the logic chart

Subroutine CHART and several lesser subprograms print a logic chart conforming to the specifications just described. The chart is constructed from the list structural representation created for an arbitrary input Coursewriter II course as discussed in Chapter III. This entire list structure is leafed through cell by cell and list by list until the end is reached.

In order to print separately the logic associated with each instruction, every pr encountered in the list structure (except at the beginning) is stored in sequence order on working list LW(8) by subroutine PR. List LW(8) is popped up to retrieve the next pr to be printed whenever the end of the instructions associated with the previous pr is reached. When LW(8) becomes empty, labels list LW(7) is searched for any labels that have not yet been printed. These isolated labels and their associated instructions are either not branched to, or else are branched to only through return registers.

When each label and each list with a reference counter exceeding 3 is printed, the header address is stored on a memory or junk list. The junk list is searched before printing in order to avoid duplication of output, and to provide for the path merger specification.

Subroutine CHART maintains, prints, and updates 16 vertical channels across the page. The channels are numbered 1 to 16 from left to right,

and printing of the list structure begins in channel 9. According to the chart specifications, other channels become occupied as branches develop, and printing of a channel terminates when a merge or unconditional branch develops.

Channel conditions are updated as the chart is printed, one horizontal line at a time. Two computer words store the contents of each channel for printing, in vector CHANEL of length 32 words, and the entire chart is printed in format (X,16(2A4)). After each line is printed, the contents of all channels are set to blanks.

The utilization of each of the 16 channels is coded in terms of one of 9 distinct conditions, numbered 1 to 9, which determine the state or condition of the channel, and what is to be printed. A complete list of these 9 conditions, and permissible subsequent conditions for each, is given in the Chart Channel Condition Code of Figure 4.1.

An integer condition vector COND, of length 16, stores the current condition code for each of the 16 channels. Initially, all channels are free except channel 9, so that COND(1) to COND(16) are each 6, except COND(9) has the value of 8. This permits the printing of a vertical bar in channel 9 to start the chart. As branches develop, the conditions of the other channels change through 6, (perhaps 7), and 8, to condition 1, indicating that the channel is in use. When a merge or unconditional branch develops to terminate the printing of a channel, the channel is first put in a state of suspension, codes 4 and 5, before becoming free. This assures that a channel is blank for the printing of one or two lines before it can come back into use. Thus a typical sequence of condition codes that a channel may assume, as a series of lines is printed, might

Code	Meaning	Permissible Subsequent Conditions
1	channel is in use	1, 2, 3, 4, 9
2	attempt to branch	1, 2, 4
3	prepare to suspend	4
4	begin suspension	5
5	continue suspension	6
6	available or free	6, 7, 8
7	prepare for use after branch to right	8
8	prepare for use; print vertical bar	1
9	print merge after label before suspension	3, 4

Figure 4.1 Chart Channel Condition Code

be 6,8,1,4,5,6 as the condition changes from free to busy to suspended to free again.

Entirely different portions of the list structure may be printed in different channels of the chart at the same time. Relationships between channels come about only through branching or merging. Subroutine CHART stores the address of the list structural cell which each channel has printed in the integer vector CELL, of length 16. When a channel is in normal use (for example, printing a series of minor instructions), the cell for the channel is advanced one to the right in the list structure after each line is printed.

Before a new line is printed, each channel is examined in turn from left to right in order to decide what should be printed, and what the subsequent cell and condition code should be for the channel. For a condition code of 1, the identification field (ID) of the current cell is first examined. Subsequent action depends on the value of the ID,

in accordance with the Cell ID Code of Figure 3.7.

If the ID is 2 or more, a list header has been encountered. The junk list is searched to see if the list has been previously printed. If it has, then the condition code is changed to 3 (or 4 for labels) to provide for merger and suspension of further printing in the channel. The list reference number (or label) is then stored in CHANEL for printing. On the other hand, if the list has not previously been printed, then its address is stored on the junk list and the condition code is not changed.

For a condition code of 1 and a current cell ID of 1, a merger sublist has been encountered. Accordingly, a vertical bar is stored in CHANEL for printing, and CELL is changed to the address of the header of the sublist. If the ID is -2, -3, or -4, an immediate attempt is made to indicate a branch to another channel. If this is not possible, the channel condition code is changed to 2 and a vertical bar is stored for printing to indicate continuation of the channel in accordance with the specifications.

For any other cell ID and a condition code of 1, the cell contents is printed in accordance with the ID as in Figure 3.7. For example, if the ID is 0, the contents are printed as a single word of characters, whereas if the ID is -5, the contents is an integer which must first be changed to characters using subroutine HOLLER, and indentation and a possible minus sign must be provided for in printing. In any case CELL is advanced to the next appropriate cell, and an immediate branch is attempted if the ID of the new cell so dictates.

When a channel has a condition code of 2, branching is attempted by

calling function NUCHAN to determine the channel that may be branched to (if any). NUCHAN examines the conditions of neighbouring channels to either side, and selects the side on which there is more room (left or right depending which side has the most consecutive free channels adjacent to the channel in question). NUCHAN then selects as a branch channel one that is about one third of the way to the first busy channel or to the edge of the paper. This seems to provide adequate spacing across the page, in an esthetically pleasing manner, for most purposes. If no branch channel is available, the condition code for the current channel remains at 2. Otherwise, subroutine ARTIST is called to draw strokes (---) to the branch channel, and to print T-OUT or MIS-M if the cell ID is -3 or -4 respectively.

A channel condition code of 3 is a pre-suspension condition used with lists that are not representing labels. With these lists, the reference number (header address) is to be printed first, followed by the instruction. This requires one more line than for labels, which are unique and do not require a reference number. Accordingly, when the channel condition code is 3, the cell contents are stored for printing and the new condition code is set to 4.

When the current channel condition code is 4 or 5, the channel is in suspension prior to becoming available, and the new condition code is set to 5 or 6 respectively. This means that at least one blank position will be printed after a channel has been in use and before it becomes available for use again.

A channel condition code of 6 indicates that the channel is available or free for use. The channel remains available until selected by

NUCHAN to be the branch destination for some other nearby channel. The condition code then changes to 8, or else 7 for a destination channel that is to the right of the branching channel. A condition code of 7 changes to 8 before a line is printed, since channels are examined in turn from left to right before a line is printed.

A condition code of 8 designates a channel that is about to come into use. A vertical bar (|) is stored in CHANEL for printing, and the condition code is changed to 1.

Condition code 9 is sometimes used when a label is encountered that has previously been printed. Ordinarily the channel code would be set to 4, but if the label is followed immediately by another label or instruction with a reference number, then code 9 is used to delay suspension of the channel until the label or instruction has been printed. Accordingly, CELL is advanced and the condition code is changed to 3 or 4. This condition provides for the printing of an important instruction immediately following a label even if the instruction has been printed before.

3. Chart Example

A listing of a short series of Coursewriter statements is given in Figure 4.2. This course listing is augmented on the left by label-sequence numbers and CL2 reference numbers (list header addresses). For example, the statement un has label-sequence number LABEL1-6 and CL2 reference number 129. This augmented listing is printed by the CL2 main program as the input Coursewriter cards are being read and the list structure is being created.

CL2 reference numbers form two separate non-decreasing sequences,

one for pr instructions and the other for any other numbered instructions. This means the augmented listing can be quickly scanned in order to locate any numbered instruction. As an additional scanning aid, each label, pr and ep is indicated in successive columns from the left margin of the listing.

```
          PR          19          1  PR
                                  2  LR   LAB3¬/RR2
                                  3  DT1
                                  4  BR   LAB2¬/S1¬/0
                      57          5  AA
                                  6  DT2
                                  7  LD   -12¬/C14
                EP    70          8  EP
                                  9  DT3
                      78         10  NX
                                 11  DT4
                      87         12  AB
                      90         13  DT5
LABEL1                         LABEL1
                                  1  DT6
                     106          2  NX
                                  3  DT7
                                  4  LD   C14¬/C15
                                  5  BR   LAB2
                     129          6  UN
                                  7  DT8
LAB2                            LAB2
                                  1  DT9
LAB3                            LAB3
                                  1  DTA
                     152          2  EA
                                  3  DTB
                                  4  BR   RR2¬/C1¬/LE¬/10
                                  5  DTC
                                  6  TR   2¬/SECOND
```

Figure 4.2

Augmented Course Listing of Logic Chart Example

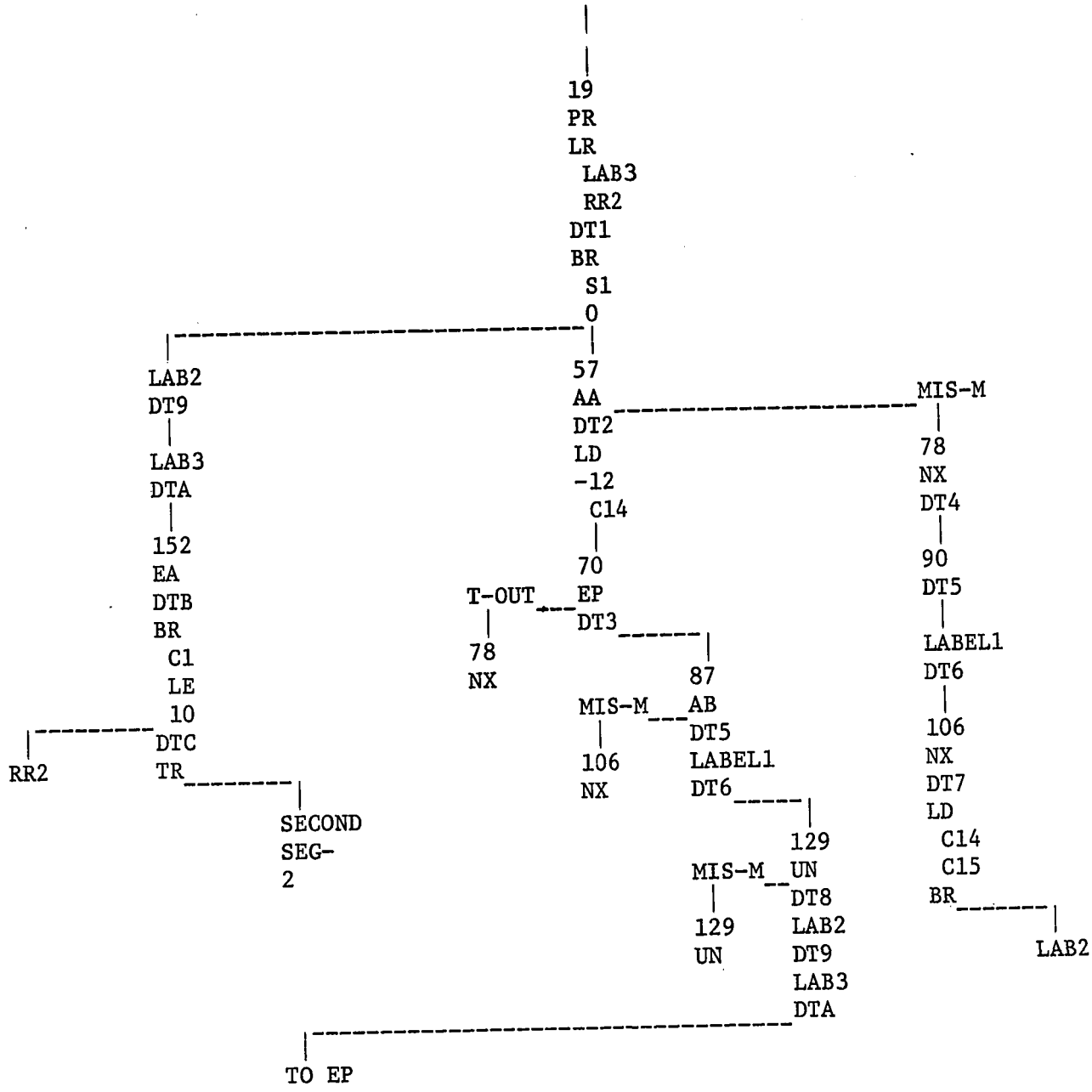

Figure 4.3 Logic Chart Example

The logic chart is shown in Figure 4.3. Sequential execution of the first four instructions is indicated by their consecutive vertical ordering. The pr instruction, since it is represented by a list header, is preceded above by its reference number 19, which is the list header address. The items or parameters of lr (LAB3 and RR2) are indented and printed below the instruction. Similarly the parameters of br (S1 and 0) are indented and printed below it. The destination of this conditional explicit branch is denoted by strokes to label LAB2 which heads a channel to the left. If the condition is false (that is, if switch S1 does not have the value 0), then continuation of sequential execution is indicated, and instruction aa with reference number 57 is executed next.

When 57 aa is executed, a match or mismatch results. In case of mismatch, a branch to 78 nx is indicated by strokes to a channel at the right. Otherwise, if there is a match on the aa, sequential execution continues and dt2 is executed, followed by ld with parameters -12 and C14.

Then ep with reference number 70 is executed, and if the response is not in time, an implicit branch to 78 nx is indicated. This time-out channel merges with 78 nx in the right hand channel. Otherwise, if 70 ep receives a response in time, sequential execution continues as dt3 is executed. This is followed by an unconditional implicit branch to 87 ab.

In case of mismatch on the ab, an implicit branch takes place to 106 nx which has already been printed in a channel to the right. Otherwise, if a response match takes place when the ab is executed, then sequential execution of dt5, LABEL1, and dt6 is indicated, followed by an unconditional implicit branch to 129 un. The instructions dt5, LABEL1 and dt6 occur also on the path in the channel at the right, but the outcome is

different on each path, so the two paths through the same instructions are shown separately. This duplication of instructions on more than one path leads to more complex program logic than what is ordinarily practised, so such occurrences are rare.

However, in this example another such duplication occurs with label LAB2 and the three following instructions. These instructions occur both in the left channel, and also below 129 un. It is important to note that when the rightmost channel branches to LAB2, this branch is to the point where LAB2 is preceded above by a vertical bar, namely the left channel.

The channel headed by 129 un ends with an unconditional implicit branch "TO EP". The program has not determined which ep was the last one executed for this path.

B. Systematic Path Tracing

While the logic chart shows diagrammatically the entire logical structure of a course, it is deficient in being static rather than dynamic. For example, conditional branches may be charted for which the truth evaluation of the conditional expression depends upon the path taken up to that point. A program that systematically traces all possible paths that a student could take through a course would therefore provide additional useful information about the dynamic logic of the course.

A problem that arises in attempting to trace possible paths through a course is the fact that any path is in part determined by student responses. While it is possible to consider tracing paths generated by

all possible response combinations, the number of paths so generated quickly becomes impractically large even for a relatively short course. It was therefore decided to trace only those paths generated on the assumption of response consistency, which means that the same response will be made if the same response-dependent instruction is executed more than once. The response consistency assumption is defined more precisely below.

Another path tracing requirement is that paths be traced in a systematic fashion. This is necessary both in order to avoid retracing the same path, and to enable the user to locate particular paths of interest from a large set of paths that may be output by the program. For instance, an author may find it useful to determine what happens when a student fails to respond in time, especially if no special provision has been made for such a contingency.

1. Path Tracing Specifications

(a) Execution. In order that paths be correctly traced, the execution of all instructions that affect branching must be simulated. The truth of the conditional expression in an explicit conditional branch must be evaluated, and switches, counters, and return registers must be updated as required. Unfortunately, it is not easily possible to simulate the execution of functions that affect branching, since functions are not written in Coursewriter. Therefore functions are not executed.

(b) Printing. In order to provide the minimum of printed output consistent with accurate recording of the path of execution, only labels or numbered instructions are to be printed. Since all branches

involve labels or instructions with reference numbers, the printing of these as they are executed provides an unambiguous record of the path traced. If an ep is in time, T-OUT is not printed, and if a match is selected during response analysis, MIS-M is not printed.

(c) Systematic tracing. Paths are traced systematically in a very definite order. The first path times out on every response request and mismatches on every response analysis instruction (aa, ca, wa, wb, cb, and ab; matches on un are determined entirely by internal counters). When a path terminates, the next path is defined by changing the last executed mismatch or time-out to a match or time-in. However, since a timed-out response cannot be matched, a response request instruction is placed in time before any subsequent answer mismatches are changed to matches. Also, once a response request (ep) instruction has been placed in time, it is not allowed to time out on subsequent paths. With these conventions, a systematic series of paths may be traced until, on the last path, every response-dependent instruction is in time or matched.

(d) Response consistency. If a response-dependent instruction (response request or response analysis) is executed more than once along the same route in a path, the same response condition will apply each time. For example, if an ep is in time and a subsequent ca is matched, and if the same ep and ca are executed again on the same path, they will still be in time and matched respectively.

If the assumption of response consistency is viewed as a student response model, the students so defined are unrealistically characterized by both a lack of forgetfulness and an inability to learn

anything new. The purpose of the path tracing is, however, to illustrate features of the dynamic logic of a course, and the response consistency convention reduces the number of redundant paths traced usually without loss of important features of the instructional logic. Much more will be said about student response models in the next section.

(e) Path termination. Paths may end in one of five distinct ways - an infinite loop, transfer to another segment, termination of course, branch to a non-existent label, or branch to an empty return register. Most of these types of path termination are represented in the output of the path tracing to be described.

2. Path Tracing Program

In order to trace paths through a course, it is essential to simulate the execution of the instructions that affect branching. Counters, switches, and return registers must be represented, and the representations must be updated whenever a Coursewriter II instruction that changes their values is encountered. The truth value of a conditional expression in a conditional explicit branch must be evaluated to decide whether to branch or not. The last executed response request (ep) instruction must be stored, since it may be the destination of a branch, and the operation of the two un counters must be simulated.

The 31 counters, 528 switches (32 standard switches and 1 switch for each of the 16 bits of each of the 31 counters) and 6 return registers are each represented by a separate integer variable. Whenever a Coursewriter II instruction is encountered that changes the values of counters, switches or return registers (instructions ad, sb, mp, dv, lr, and ld,

except load buffer), the TRACE subprogram calls subroutine EXCUTE to update the appropriate representation.

Subroutine EXCUTE first determines whether it is a switch, counter, or return register that is to be changed. If it is a switch, function ISWICH is called to determine which switch, while if it is a counter, function ICOUNT determines which counter. Arithmetic operations with integers and counters are performed as required, and the new value is stored in the appropriate simulated switch or counter. Execution of the load register (lr) instruction is simulated by storing the label address in the appropriate return register representation. The value of the simulated UN2 counter (represented by integer variable IUN2) is also stored whenever a label is loaded into a return register.

When the TRACE subroutine encounters a conditional explicit branch, logical function IVAL is called to evaluate the truth value of the conditional expression. For this evaluation, ISWICH and ICOUNT are called to determine which counters and switches are involved, and then the values of the appropriate simulated counters and switches are used to make the comparison that determines the logical outcome.

The TRACE subroutine stores the address of the last executed ep instruction, and increments the simulated UN2 counter whenever the same ep is executed again. When a un instruction is encountered, match or mismatch is determined by comparison of the simulated UN2 counter with the simulated UN1 counter which is stored in a cell in the list structure. When a branch is encountered to a return register, the address of the appropriate label is retrieved from the simulated return register, and the value of the simulated UN2 counter is restored.

A path is traced through a course by advancing cell by cell through its list structural representation. Each time a list header is encountered, its address is stored on a list named MEMORY. The MEMORY is also searched to see if the list address is already stored there, in which case the same instruction is being re-executed. Program loops are detected in this manner. Tracing of a path terminates when a program loop has been executed ten times (or a specified number of times, on the assumption that an endless loop has been encountered). Tracing will also terminate if the end of a list is reached and no branch or merge has been specified.

The MEMORY list stores an account of the path being traced, and is used by subroutine PATH to print a trace of the labels and numbered instructions whose execution has been simulated. Subroutine PATH prints an instruction preceded by its reference number (list header address). For response-dependent instructions, T-OUT or MIS-M is printed if a time-out or mismatch response has been selected. The word LOOP is printed at the end of paths that terminate in an endless loop.

Whenever a path decision depends on a student response, the convention of systematic tracing and the assumption of response consistency are used to make the decision. On the first path, time-out or mismatch responses are made at each response-dependent decision point (referred to as a "door"). In other words, all doors are open on the first path, but the last open door is remembered. When the path terminates, the last open door is stored on a list that records the addresses of all doors that are to be shut.

When subsequent paths are systematically traced, doors that are stored on the record list, in order, are not permitted to be opened. At

the termination of each path, the last open door is added to the record list of doors that are shut, and any closed doors after the last open door are removed from the list. Finally, on the last path, all doors have been shut, and in-time or match responses are made at each response-dependent decision point.

An ep instruction is placed in time before the subsequent response analysis instructions are permitted to match. Then the address of the ep is stored so that it is not permitted to time out on subsequent paths. When any response-dependent instruction is re-executed, the same response is made again.

Along each path, a count is maintained of the total number of statements executed on the path, and this information is printed at the termination of the path.

3. Path Tracing Example

A listing of a short series of Coursewriter II statements for a path tracing example is given in Figure 4.4. The listing is augmented on the left by label sequence numbers, reference numbers (list header addresses), and indication of the location of each ep, pr, and label.

Figure 4.5 shows the paths traced through the series of instructions listed in Figure 4.4. In accordance with the specifications, only labels and numbered instructions are printed, but the execution of all instructions that affect branching has been simulated.

The first path times out and mismatches at every opportunity. As counter C1 is being incremented, the path times out four times on instruction 47 EP before the conditional expression of the conditional branch

is false, and then execution branches to 21 PR. Response consistency is evident as the path times out continually on 134 EP and terminates in an endless loop.

PATH NO. 2 is determined from PATH NO. 1 by changing 134 EP from time-out to in time. This leads to a new route and mismatches on 152 CA and 159 FN2 before PATH NO. 2 terminates. PATH NO. 3 is then determined by matching on 159 FN2, and PATH NO. 4 by matching on 152 CA. PATH NO. 5 first puts 47 EP in time before matching on 73 CA, 80 CB, or 100 WA. PATH NO. 6 ends in a loop by consistently giving wrong answer 100 WA to response request 47 EP.

```
        PR          19          1  PR
                                2  LD  0¬/C1
                                3  LD  0¬/S1
            EP      47          4  EP
                    54          5  NX
                                6  AD  1¬/C1
                                7  BR  RE¬/C1¬/LE¬/3
                                8  BR  PR1
                    73          9  CA
                    80         10  CB
                    83         11  LR  LABEL¬/RR0
                   100         12  WA
                   109         13  UN
                   118         14  UN
        PR          21         15  PR
            EP     134         16  EP
                   141         17  NX
                               18  BR  RE
                   152         19  CA
                   159         20  FN2
                               21  TR  001¬/ONE
                   175         22  UN
                               23  BR  RR0¬/S1¬/0
                               24  EA
LABEL                         LABEL
                                1  LD  1¬/S1
                                2  BR  PR0
        PR          23          3  PR
```

Figure 4.4

Augmented Listing of Path Tracing Example

```
PATH NO.   1          51 STATEMENTS EXECUTED
      19 PR    47 EP T-OUT    54 NX    47 EP T-OUT    54 NX    47 EP T-OUT
      54 NX    47 EP T-OUT    54 NX    47 EP T-OUT    54 NX    21 PR
      134 EP T-OUT   141 NX    134 EP    LOOP

PATH NO.   2          27 STATEMENTS EXECUTED
      19 PR    47 EP T-OUT    54 NX    47 EP T-OUT    54 NX    47 EP T-OUT
      54 NX    47 EP T-OUT    54 NX    21 PR    134 EP    152 CA MIS-M
      159 FN2 MIS-M    175 UN     RRO

PATH NO.   3          28 STATEMENTS EXECUTED
      19 PR    47 EP T-OUT    54 NX    47 EP T-OUT    54 NX    47 EP T-OUT
      54 NX    47 EP T-OUT    54 NX    21 PR    134 EP    152 CA MIS-M
      159 FN2    ONE

PATH NO.   4          24 STATEMENTS EXECUTED
      19 PR    47 EP T-OUT    54 NX    47 EP T-OUT    54 NX    47 EP T-OUT
      54 NX    47 EP T-OUT    54 NX    21 PR    134 EP    152 CA    23 PR

PATH NO.   5          68 STATEMENTS EXECUTED
      19 PR    47 EP    73 CA MIS-M    80 CB MIS-M    100 WA MIS-M
      109 UN    47 EP    73 CA MIS-M    80 CB MIS-M    100 WA MIS-M
      109 UN MIS-M    118 UN    47 EP    LOOP

PATH NO.   6          43 STATEMENTS EXECUTED
      19 PR    47 EP    73 CA MIS-M    80 CB MIS-M    100 WA    47 EP
      LOOP

PATH NO.   7          67 STATEMENTS EXECUTED
      19 PR    47 EP    73 CA MIS-M    80 CB    83 LR    21 PR    134 EP
      152 CA MIS-M    159 FN2 MIS-M    175 UN    LABEL    21 PR    134 EP
      152 CA MIS-M    159 FN2 MIS-M    175 UN    134 EP    LOOP

PATH NO.   8          15 STATEMENTS EXECUTED
      19 PR    47 EP    73 CA MIS-M    80 CB    83 LR    21 PR    134 EP
      152 CA MIS-M    159 FN2    ONE

PATH NO.   9          11 STATEMENTS EXECUTED
      19 PR    47 EP    73 CA MIS-M    80 CB    83 LR    21 PR    134 EP
      152 CA    23 PR

PATH NO.   10         66 STATEMENTS EXECUTED
      19 PR    47 EP    73 CA    83 LR    21 PR    134 EP    152 CA MIS-M
      159 FN2 MIS-M    175 UN    LABEL    21 PR    134 EP    152 CA MIS-M
      159 FN2 MIS-M    175 UN    134 EP    LOOP

PATH NO.   11         14 STATEMENTS EXECUTED
      19 PR    47 EP    73 CA    83 LR    21 PR    134 EP    152 CA MIS-M
      159 FN2    ONE

PATH NO.   12         10 STATEMENTS EXECUTED
      19 PR    47 EP    73 CA    83 LR    21 PR    134 EP    152 CA
      23 PR
```

Figure 4.5 Path Tracing Example

When PATH No. 7 matches on 80 CB, a route again develops to 134 EP which is not permitted to time out since it was already placed in time on PATH NO. 2. The response analysis instructions 152 CA and 159 FN2 following 134 EP are however again permitted to mismatch, since the conditions of counters, switches, and return registers may be different. This is indeed the case in this example, since LABEL was loaded into RR0 on PATH NO. 7 but not on PATH NO. 2.

The remaining paths are traced in turn until at last on PATH NO. 12 all EP's are in time and no response mismatches occur. This is also the shortest path in this example.

C. Student Response Simulation

A major objective of this thesis is to show how the computer may be used as a means of representing or simulating student response models. A student model can then interact with a course representation to simulate both the instructional and learning processes. So far, it has been shown how the decision structure of a course or logical sequence of instructions as described in Coursewriter II language, can be simulated. In this section, a framework for the representation of student models is described, and two simplified examples of student response models and their interaction with a course are illustrated.

1. A Framework for Student Models

An important step in computer simulation is the development and testing of adequate models of the process being simulated. Here it is the instructional and learning aspects of a computer-assisted instructional (CAI) setting that are being simulated. The development of models

is considered in this chapter and the previous chapters, and the adequacy of the models is considered in the next chapter.

Student responses influence the instructional and learning process of CAI by determining the decisions taken at response-dependent decision points (nodes) in the course. In Coursewriter II, nodes are generated by execution of the response request instruction (ep in time or timed-out) and some of the response analysis instructions (aa, ca, wa, wb, cb, ab match or mis-match). A student response model must specify the decision to be taken at each such node. The instructional logic of a course, in conjunction with the student model, determines the path followed by each simulated student through the course.

An evaluation of the adequacy of a student response model can be made by detailed comparison of the paths followed by real and simulated students through a course. However, it is frequently desirable to compare real and simulated students on the basis of generated variables which summarize aspects of the paths. For instance, the path length in terms of the number of instructions executed in a course can be compared for real and simulated students. Therefore a facility for generation of simulation variables which are a function of the paths followed by simulated students is a requirement of a framework for expressing student simulation models.

To trace the paths followed by simulated students through a course, subroutine STRACE advances cell by cell through the list structural representation in a manner similar to the TRACE subroutine described in the previous section. The assumption of response consistency is replaced by a call to a special MODEL subroutine whenever a response-dependent node is encountered.

Subroutine MODEL provides the framework for representing student response models, as well as a facility for the generation of simulation variables. In order to provide for both of these functions, MODEL is called at response-dependent decision points so that a simulated response may be made, and MODEL is also called whenever any instruction is executed so that path-dependent simulation variables may be generated. Indication of the location of the node or instruction being executed is provided.

The six arguments of subroutine MODEL are:

N - student number

IX - integer vector of simulation variables that may be created for each student

IREF - the reference number (list header address) of the last label or numbered instruction

ICELL - the MOSLIP address of the current instruction

NODE - logical variable that is true whenever a response-dependent node or decision point is encountered, otherwise false.

BRANCH - logical variable that should be set false for a student to be in time or match at a response-dependent node; otherwise true (time-out or mismatch) is assumed.

The MODEL subroutine is called under the two different circumstances already mentioned:

(a) MODEL is called whenever a node is generated, immediately after execution of a response-dependent instruction. In this case, NODE is true and IREF is the reference number (list header address) of the node. A decision must be made, on the basis of the student model, whether or not to set BRANCH to false. If BRANCH is not set

to false, then true is assumed and the simulated student will time-out or mismatch at the node.

(b) Subroutine MODEL is also called whenever any instruction is executed. In this case, NODE is false and ICELL is the MOSLIP address of the instruction. No decision must be made about the truth value of BRANCH. Instead, information about the instruction may be used to generate any desired simulation variables.

Thus subroutine MODEL provides a framework for the Fortran coding of student response models.

2. A Random Response Model

This first student simulation example shows how a very simple stochastic response model may be coded. At each response-dependent decision point (node) in the course, a pseudorandom number uniformly distributed between 0 and 1 is generated. If the random number is less than 0.5, a match or in time response is made, otherwise a mismatch or time-out response is selected. Thus each simulated student has equal probability of responding in time (matching) or timing-out (mismatching) at each node.

The Fortran coding of this random response model as represented by subroutine MODEL is shown below:

```
      SUBROUTINE MODEL (N,IX,IREF,ICELL,NODE,BRANCH)
C     MODEL 1 - RANDOM RESPONSE MODEL
      INTEGER IX(1),IR/5093/
      LOGICAL NODE,BRANCH
      IF(NODE) GO TO 99
      IX(1)=IX(1)+1
      RETURN
99    CONTINUE
      CALL RANDU(IR,IY,R)
      IR=IY
      IF(R.LT.0.5) BRANCH=.FALSE.
      RETURN
      END
```

```
STUDENT      1        20 STATEMENTS EXECUTED
     19 PR    47 EP    73 CA MIS-M    80 CB    83 LR    21 PR
     134 EP T-OUT    141 NX    134 EP T-OUT    141 NX    134 EP T-OUT
     141 NX    134 EP    152 CA    23 PR
     VARIABLE    1
                20


STUDENT      2        26 STATEMENTS EXECUTED
     19 PR    47 EP T-OUT    54 NX    47 EP T-OUT    54 NX    47 EP T-OUT
     54 NX    47 EP    73 CA MIS-M    80 CB    83 LR    21 PR
     134 EP T-OUT    141 NX    134 EP    152 CA    23 PR
     VARIABLE    1
                26


STUDENT      3        21 STATEMENTS EXECUTED
     19 PR    47 EP    73 CA MIS-M    80 CB MIS-M    100 WA MIS-M
     109 UN    47 EP    73 CA    83 LR    21 PR    134 EP T-OUT    141 NX
     134 EP T-OUT    141 NX    134 EP    152 CA    23 PR
     VARIABLE    1
                21


STUDENT      4        24 STATEMENTS EXECUTED
     19 PR    47 EP T-OUT    54 NX    47 EP T-OUT    54 NX    47 EP T-OUT
     54 NX    47 EP T-OUT    54 NX    21 PR    134 EP    152 CA    23 PR
     VARIABLE    1
                24


STUDENT      5        46 STATEMENTS EXECUTED
     19 PR    47 EP    73 CA MIS-M    80 CB MIS-M    100 WA MIS-M
     109 UN    47 EP T-OUT    54 NX    47 EP    73 CA    83 LR    21 PR
     134 EP T-OUT    141 NX    134 EP T-OUT    141 NX    134 EP T-OUT
     141 NX    134 EP T-OUT    141 NX    134 EP T-OUT    141 NX
     134 EP T-OUT    141 NX    134 EP T-OUT    141 NX    134 EP T-OUT
     141 NX    134 EP T-OUT    141 NX    134 EP    152 CA    23 PR
     VARIABLE    1
                46
```

Figure 4.6

Simulation Output for Random Response Model

The two separate circumstances under which subroutine MODEL is called are clearly evident in the coding of the above random response model. If a node is encountered, execution transfers to 99. Otherwise variable IX(1) is incremented by 1 and execution returns to the calling program (STRACE). Since all variables are initially set to zero for each student, and since NODE is false whenever MODEL is called at the execution of an instruction, variable IX(1) will add up to be the total number of instructions executed for each student. This agrees with the printed simulation output for the random response model (Figure 4.6), as applied to interact with the course sequence of Figure 4.4. For example, 20 statements were executed for the first simulated student, and the value of variable 1 is also 20.

Whenever MODEL is called at a response-dependent node, NODE is true and subroutine RANDU (from the IBM System/360 Scientific Subroutine Package, 1968) generates a pseudorandom number R uniformly distributed between 0 and 1. The integer 5093 is the starting point of the sequence of pseudorandom numbers. If R is less than 0.5, BRANCH is set to false (the response is a match or in time), otherwise true is assumed and the response is a mismatch or time-out.

In this way, the random response model provides for simulated student responses whenever a response-dependent decision point is encountered in a CAI course. However, this simple model is unlikely to give an accurate simulation of the response of real students.

3. A Two-Group Model

This is a second example of a student simulation model for use with

the course sequence of Figure 4.4. In this model there are two groups of simulated students, referred to as HIGH and LOW, with four students in each group. Probabilities of being in time or matching are specified for each group at each of the seven response-dependent nodes of the example. The fourteen variables representing frequencies of path selection (in time or match and time-out or mismatch frequencies) at each node are generated for each student. Simulated students 1 to 4 are in the HIGH group, while students 5 to 8 are in the LOW group. Here is the Fortran coding for this second model.

```
      SUBROUTINE MODEL(N,IX,IREF,ICELL,NODE,BRANCH)
C     MODEL 2
      INTEGER IX(1),IR/5093/
      LOGICAL NODE,BRANCH
      INTEGER NODES(7)/47,73,80,100,134,152,159/
      REAL HIGH(7)/.90,.70,.70,.20,.80,.60,.90/
      REAL LOW(7)/.45,.35,.35,.40,.40,.30,.45/
      IF(.NOT.NODE) RETURN
      DO 100 I=1,7
      IF(IREF.EQ.NODES(I)) GO TO 102
100   CONTINUE
      WRITE(6,101) IREF
101   FORMAT('UNEXPECTED NODE'I5,'ENCOUNTERED')
      STOP
102   CALL RANDU(IR,IY,R)
      IR=IY
      IF(N.LE.4.AND.R.LT.HIGH(I)) GO TO 103
      IF(N.GT.4.AND.R.LT.LOW(I)) GO TO 103
      IX(2*I)=IX(2*I)+1
      RETURN
103   IX(2*I-1)=IX(2*I-1)+1
      BRANCH=.FALSE.
      RETURN
      END
```

The array NODES contains the reference numbers of the 7 response-dependent nodes in the example: 47 EP, 73 CA, 80 CB, 100 WA, 134 EP, 152 CA, and 159 FN2. Arrays HIGH and LOW contain the in-time or match probabilities of the HIGH and LOW groups at each of the 7 nodes. The HIGH group is given twice the probability of the LOW group for being in

time or matching at each node except node 100 WA, where the LOW group has twice the probability of matching as the HIGH group. The values of the probabilities could be estimated by the course author from his familiarity with the course and with the students being simulated. Alternatively, if the intention of the simulation is to test a learning theory, branching decisions should be derived from the theory. For this fictitious example, the probabilities HIGH and LOW were selected fictitiously.

This second MODEL does not make use of information about instructions encountered, so that execution returns to the calling program if NODE is not true. The subscript I of a node is identified in the DO 100 loop by matching its reference number with the stored array NODES. An error message is printed if no match is found. Otherwise RANDU generates a random number R uniformly distributed between 0 and 1. If R is less than the stored probability (HIGH(I) for students 1 to 4 and LOW(I) for students 5 to 8) of being in time or matching at the specific node, execution transfers to 103 and BRANCH is set to false. Otherwise BRANCH is true.

The variable representing the in-time (match) counter for the node (IX(2*I-1)) or the time-out (mismatch) counter (IX(2*I)) is also incremented accordingly. For example, with IREF equal to 47 (node 1), variable IX(1) is incremented if R is less than HIGH(I) or LOW(I), otherwise variable IX(2), the time-out counter for node 1, is incremented. The result is that, for each student, the odd-numbered variables IX(1) to IX(13) are the in-time (match) frequencies, while even-numbered variables IX(2) to IX(14) are the time-out (mismatch) frequencies for nodes 1 to 7.

Simulation output for the two group model (Figure 4.7) shows a predictable tendency for the students in the LOW group to have longer

```
STUDENT    1              15 STATEMENTS EXECUTED
      19 PR    47 EP     73 CA MIS-M     80 CB     83 LR     21 PR     134 EP
      152 CA MIS-M     159 FN2     ONE
      VARIABLE   1   2   3   4   5   6   7   8   9   10   11   12   13   14
                 1   0   0   1   1   0   0   0   1    0    0    1    1    0

STUDENT    2              14 STATEMENTS EXECUTED
      19 PR    47 EP     73 CA     83 LR     21 PR     134 EP     152 CA MIS-M
      159 FN2     ONE
      VARIABLE   1   2   3   4   5   6   7   8   9   10   11   12   13   14
                 1   0   1   0   0   0   0   0   1    0    0    1    1    0

STUDENT    3              14 STATEMENTS EXECUTED
      19 PR    47 EP     73 CA     83 LR     21 PR     134 EP     152 CA MIS-M
      159 FN2     ONE
      VARIABLE   1   2   3   4   5   6   7   8   9   10   11   12   13   14
                 1   0   1   0   0   0   0   0   1    0    0    1    1    0

STUDENT    4              16 STATEMENTS EXECUTED
      19 PR    47 EP     73 CA     83 LR     21 PR     134 EP T-OUT     141 NX
      134 EP T-OUT     141 NX     134 EP     152 CA     23 PR
      VARIABLE   1   2   3   4   5   6   7   8   9   10   11   12   13   14
                 1   0   1   0   0   0   0   0   1    2    1    0    0    0

STUDENT    5              11 STATEMENTS EXECUTED
      19 PR    47 EP     73 CA MIS-M     80 CB     83 LR     21 PR     134 EP
      152 CA     23 PR
      VARIABLE   1   2   3   4   5   6   7   8   9   10   11   12   13   14
                 1   0   0   1   1   0   0   0   1    0    1    0    0    0

STUDENT    6              24 STATEMENTS EXECUTED
      19 PR    47 EP T-OUT     54 NX     47 EP T-OUT     54 NX     47 EP T-OUT
      54 NX    47 EP T-OUT     54 NX     21 PR     134 EP     152 CA     23 PR
      VARIABLE   1   2   3   4   5   6   7   8   9   10   11   12   13   14
                 0   4   0   0   0   0   0   0   1    0    1    0    0    0

STUDENT    7              46 STATEMENTS EXECUTED
      19 PR    47 EP     73 CA MIS-M     80 CB MIS-M     100 WA MIS-M
      109 UN     47 EP T-OUT     54 NX     47 EP     73 CA     83 LR     21 PR
      134 EP T-OUT     141 NX     134 EP T-OUT     141 NX     134 EP T-OUT
      141 NX     134 EP T-OUT     141 NX     134 EP T-OUT     141 NX
      134 EP T-OUT     141 NX     134 EP T-OUT     141 NX     134 EP T-OUT
      141 NX     134 EP T-OUT     141 NX     134 EP     152 CA     23 PR
      VARIABLE   1   2   3   4   5   6   7   8   9   10   11   12   13   14
                 2   1   1   1   0   1   0   1   1    9    1    0    0    0

STUDENT    8              24 STATEMENTS EXECUTED
      19 PR    47 EP T-OUT     54 NX     47 EP T-OUT     54 NX     47 EP T-OUT
      54 NX    47 EP T-OUT     54 NX     21 PR     134 EP     152 CA     23 PR
      VARIABLE   1   2   3   4   5   6   7   8   9   10   11   12   13   14
                 0   4   0   0   0   0   0   0   1    0    1    0    0    0
```

Figure 4.7 Simulation Output for Two-Group Model

paths through the course. This second model is more specific than the random response model, in that the model provides different parameters for each response-dependent decision point in the course. The random response model could apply to any course or sequence of Coursewriter II instructions, but the two-group model just described has been tailored to meet specific requirements of the particular course.

CHAPTER V

Discussion

DISCUSSION

A. Evaluation of the Simulation

The preceding chapters have presented a simulation of the logical aspects of the interaction that takes place between computer and student in a computer-assisted instructional setting. The simulation programs developed deal with both the instructional and learning aspects of this interaction, specifically for the IBM 1500 Instructional System and the Coursewriter II language.

The programs that develop the list structural representation of a course, as described in Chapter III, and the printout of this structure in the form of a logic chart (Chapter IV, section A) deal only with instructional aspects of CAI. In contrast, the systematic path tracing and the student response model programs (Chapter IV, sections B and C) involve dynamic simulation of execution of a course.

Adequacy of the simulation is now discussed with respect to three criteria - generality, accuracy, and usefulness.

1. Generality

Although the simulation was developed in the context of the University of Alberta IBM 360/67 and the IBM 1500 Instructional System, the application of the programs is more general. The simulation is intended to input any course which meets the specifications of the Coursewriter II language, regardless of the origin or purpose of the course. However, the simulation programs developed in this thesis do not accept course input in any other CAI language, and could not easily be modified to do

so. Nevertheless, some of the ideas relating to storage of course logic in list structural form, and the specifications developed for a logic chart and path tracing could be applied to develop simulation programs for other course languages.

The simulation programs are implemented on the IBM 360/67. Because of the character manipulation operations developed for the programs, a dependency exists on the 32-bit word and the two's-complement representation of integers. Thus considerable modification would be required to make the simulation operational on a computer that doesn't have these characteristics. However, it is easy to change the program to operate with less core (for example, on a smaller 360). Dimension statements for linear arrays exist in subprograms IDIN, LLIN, LRIN, SETIND, STRIND, and INHALT. The arrays are presently dimensioned at 16000, to make 16000 MOSLIP cells available for use, and this occupies 160 K bytes of core storage. The statement CALL INITAS(16000) in the main program would also have to be altered accordingly. Of course, if the dimension statements are decreased, the programs might not be able to simulate lengthy Coursewriter courses. With 16000 MOSLIP cells, it is estimated that a typical course of up to 5000 Coursewriter II instructions can be handled.

The specifications of the logic chart (Chapter IV, section A) put some restrictions upon the type of course for which a chart can be printed. It is possible for all 16 channels to become occupied at once, and any further branching introduces an endless printing loop as each channel waits for others to clear. Since the instructions associated with each problem (pr) are printed separately, this difficulty can arise in printing the chart for a course with insufficient number of pr instructions. The

generality of the logic chart program is restricted to courses that have pr statements at convenient places.

This difficulty with the logic chart is associated with the fact that instructions following a branch to a label are printed even if the label is found much later in the course input data. This permits a multiplicity of branches to appear within close proximity to each other on the chart. The accuracy of the chart is not affected by this difficulty; the consequences are overcrowding and the possibility of an endless printing loop.

An important feature of the simulation is the separation of the instructional aspects (course material and instructional logic) from the student response or learning aspects. A user who wishes to construct and test a student response model does not have to be concerned about simulating the course, because the simulation program does this separately and automatically. Nevertheless, one may make use of information about the course (for example, the whereabouts of a simulated student in the course) in order to make decisions at response-dependent nodes and to construct simulation variables.

There is very little to restrict the generality of procedures for constructing student response models. The full algebraic power of Fortran is available to the user in the MODEL subroutine, as well as information about the course and the values of constructed simulation variables. The last example of Chapter IV illustrates the use of some of this flexibility for decision-making. However, it is necessary for the student response model to be specific enough to provide decisions at each response-dependent decision point in the course. This means that any student model to

be considered must be sufficiently definite that responses are predicted by the model whenever called for by the instructional program.

2. Accuracy

The conventions of the list structural representation developed in Chapter III are intended to define an exact simulation of the logical structure of any course written in Coursewriter II. Although it is believed that the conventions are sufficient to provide an exact representation, a proof of this assertion would be difficult, especially on account of the complexity of the implicit branching mechanism. All contingencies implied by the Course Flow Decision Table (Figure 2.1) are provided for by storing the table in coded form (Figure 3.13). However, the implementation of the conventions involves many detailed procedures, and inaccuracy could be introduced at some point.

The logic chart program (Chapter IV, section A) prints the list structure established to represent the logic of any course. A logic chart has been printed for a variety of courses, and no discrepancy from the actual logical structure has been noticed. This provides evidence in favor of the accuracy of the representation.

In contrast, the dynamic tracing of paths through the simulated logical structure of a course is easily shown to have inaccuracies under certain circumstances. There are at least two instances where the program may trace incorrect paths.

(a) In Coursewriter II, each of the 16 bits of a counter may be used as a switch at the same time, so that is possible to store an integer in a counter and refer to each bit through the corresponding switch. Simulated counters and switches are all assigned to separate storage

areas, because of the difficulties of bit manipulation. Inaccuracy in dynamic path tracing is thus introduced whenever branches depend upon the fact that counters and extra switches occupy the same storage area. For most courses, this is not a major problem.

(b) A more basic difficulty is with the use of functions (fn) that change the values of counters and switches. Functions are written in assembly language, rather than Coursewriter II, and hence are not simulated. If a course has conditional branches depending on counters and switches whose values have been set by functions, incorrect paths will be traced. This is a major problem since functions are receiving increased use on account of the many inflexibilities of the Coursewriter language.

The accuracy of student response models must be tested by comparison with real students. Because of the variability of human behavior, great accuracy in response simulation should not be expected. The development of models in this area requires special attention, and it seems likely that stochastic models should be employed (Bush & Mosteller, 1955).

3. Usefulness

Generality and accuracy are prerequisites to the usefulness of any simulation, but the stated purpose for this CAI simulation is to provide assistance for course authors. In particular, the main purpose is to aid in the development, debugging, and documentation of Coursewriter II courses for the IBM 1500.

Probably the most useful product of the thesis is the logic chart, which draws a two-dimensional picture of branches, merges, and sequential

execution of instructions in a course. The logic chart is particularly effective in portraying branches in the instructional logic. Merges are less clearly depicted, because no line is drawn to a point of merger. Instead, the user of the logic chart must search the chart on his own to find the point of merger, with the assistance of the augmented course listing for cross reference. Nevertheless, the logic chart depicts course logic in a perceptually useful form that is almost language independent.

A logic chart has been printed for a variety of courses in order to evaluate the effectiveness of the logic chart program. Crowding is sometimes a problem, because a branch to a label is printed even if the label is found much later in the course input data, so that a multiplicity of branches may develop in close proximity on the chart. This also increases the difficulty of cross referencing labels on the chart with the augmented course listing.

In addition to its usefulness in debugging and documentation of instructional logic of courses, an important application of the logic chart could be as a tool in teaching the Coursewriter II language. Because of the hidden complexity of implicit branching, it is frequently difficult for a student learning Coursewriter II to grasp the logical structure of a sequence of instructions. The logic chart has not yet been applied for this purpose, but considerable potential exists for its usefulness in this area.

The systematic path tracing program provides a detailed list of major instructions along all possible response-consistent paths in a course. This produces considerable information about the dynamic logic of a course.

However, as already mentioned, there may be errors in the accuracy of the path as constructed, particularly when functions affect counters and switches used for conditional branching. As well, the path tracing program locates more paths than what may be of interest to a course author, even with the response-consistency assumption. Path tracing would seem to be most useful for detailed logical analysis of short courses without function calls.

The usefulness of the student simulation program awaits the development of adequate student response models. Probably the main importance of this aspect of the work is the provision of a framework within which student response simulation models may be expressed, and simulation variables may be computed for testing the model against real students.

Finally, the efficiency of the simulation deserves brief mention. The simulation programs use core memory only, without recourse to peripheral storage media. The representation of a Coursewriter II course as a list structure is created in one pass, and the logic chart is printed in one traversal of the list structure. This takes about 0.7 minutes of execution time on the IBM 360/67 for each 1000 Coursewriter cards input.

Systematic path tracing and student simulation are slower because the list structure is traversed from the beginning for each path or student. In a course with about 500 instructions, 206 paths were traced in 2.40 minutes.

B. Suggested Further Research

A number of suggestions for further research stem from the evaluation of the simulation discussed in the previous section. The logic

chart could be improved by the addition of a mechanism providing for separation of output by labels as they are encountered in the course. This would solve the problem of crowding on the chart and provide better cross reference with the course listing. Perhaps also a better method could be developed for the designation of merges so that the user would not have to search the chart for the point of merger.

The systematic path tracing could be improved by providing exits for the simulation of functions. Perhaps also a more powerful scheme could be devised for the elimination of paths that are not of interest to an author. The assumption of response consistency eliminates the tracing of many similar paths, but there seems to be a greater need for selectivity in path tracing.

The development of adequate student learning models for CAI is a large area that is left untouched by the work of this thesis. Provision is made for users to code their own student models, and to tailor them to the requirements of a specific course. Methods of comparison of the paths traced by real and simulated students would need to be devised.

The simulation programs developed in this thesis for the purpose of author assistance have so far received little use by authors. Most authors develop their courses in interactive mode online with the IBM 1500, but the simulation requires entry of courses to the IBM 360/67. Thus a step is required to transfer the information, by means of cards, from the 1500 to the 360. If the simulation programs were implemented on the IBM 1500, this intermediate step and delay would not be necessary, and use could be made of graphical display devices.

REFERENCES

REFERENCES

Bobrow, D., & Raphael, B. A comparison of list-processing languages. Communications of the Association for Computing Machinery, 1964, 7, 231 - 240.

Bush, R., & Mosteller, F. Stochastic Models for Learning. New York: Wiley, 1955.

Coulson, J. (Ed.) Programmed Learning and Computer-based Instruction. New York: Wiley, 1962.

Feigenbaum, E., & Feldman, J. (Eds.) Computers and Thought. New York: McGraw-Hill, 1963.

Flathman, D. Modified Symmetric List Processor (MOSLIP). Unpublished manuscript, University of Alberta, Division of Educational Research Services, 1968.

Grubb, R. CAI - technical aspects. In R. Gerard (Ed.), Computers and Education. New York: McGraw-Hill, 1967.

Hickey, A. Computer-assisted Instruction: A Survey of the Literature. (3rd ed.) Newburyport, Mass.: Entelek, Inc., 1968.

IBM 1500 Coursewriter II Author's Guide. San Jose, California: IBM Systems Development Division, 1967.

IBM System/360 Scientific Subroutine Package. White Plains, New York: IBM Technical Publications Department, 1968.

Naylor, T., Balintfy, J., Burdick, D., & Chu, K. Computer Simulation Techniques. New York: Wiley, 1966.

Rath, G., Anderson, N., and Brainerd, R. The IBM Research Center teaching machine project. In E. Galanter (Ed.), Automatic Teaching: The State of the Art. New York: Wiley, 1959.

Sherman, P. FLOWTRACE, a computer program for flowcharting programs. Communications of the Association for Computing Machinery, 1966, 9, 845 - 854.

Silberman, H., & Filep, R. Information systems applications in education. In C. Cuadra (Ed.), Annual Review of Information Science and Technology. Vol. 3. Chicago: Encyclopedia Britannica, 1968.

Weizenbaum, J. Symmetric list processor. Communications of the Association for Computing Machinery, 1963, 6, 524 - 544.

Zinn, K. Computer technology for teaching and research on instruction. Review of Educational Research, 1967, 38, 618 - 634.

APPENDIX

Computer Program CL2

COMPUTER PROGRAM CL2

```
      COMMON LAVS,LW(10)
      INTEGER COL(80),LCURR(2),TITLE(20),BLANK/' '/
      REAL    OPCODE(20)/'EA','NX','AA','CA','WA','UN','PR','WB','CB',
     1 'AB','EP','BR','TR','AD','SB','MP','DV','LR','LD','FN'/
   91 CONTINUE
      CALL INITAS(16000)
      MINOR=0
      NPR=100
      N=0
      DO 90 I=1,2
   90 LCURR(I)=LIST(2)
      LCORSE=MADLST(LIST(3))
      CALL NEWBOT(LCORSE,NEWBOT(6,LCURR(2)))
      READ(5,88) TITLE
   88 FORMAT(20A4)
      IF(TITLE(1).EQ.TITLE(2)) STOP
      WRITE(6,99) TITLE
   99 FORMAT(1H1,37X,20A4)
      READ(5,89) ITAPE,NLOOP,ILIST,ITRACE,ICHART,NPER,NVAR,IPUNCH
   89 FORMAT(16I5)
      IF(ITAPE.EQ.0) ITAPE=5
      IF(ITAPE.NE.5) REWIND ITAPE
      IF(NLOOP.EQ.0) NLOOP=10
      DO 98 I=1,8
   98 CALL NEWBOT(MADLST(LIST(2)),LW(8))
      M=NEWBOT('TO E',NEWBOT('P   ',LW(10)))
      CALL SETIND(-6,-1,-1,M)
      LW(9)=MADLST(LW(10))
 1000 READ(ITAPE,101) COL
  101 FORMAT(80A1)
      INSTR=100
      IF(COL(1).EQ.BLANK) GO TO 103
      N=MINO(LIMTER(COL(1),12,' ',1,N),LIMTER(COL(1),12,' ',1,N))
      NEXT=NULABL(COL(1),N)
      CALL NEWTOP(11,NEWTOP(NEXT,LCURR(2)))
      LW(9)=MADLST(LW(10))
      IF(ILIST.EQ.0) WRITE(6,981) (COL(I),I=1,12),COL
  981 FORMAT(1X,12A1,25X,80A1)
      N=0
  102 IF(COL(72).EQ.BLANK.AND.LIMTER(COL(11),61,'¬*',2,M).EQ.61)GOTO1000
      READ(5,101) COL
      IF(ILIST.EQ.0) WRITE(6,104) COL
  104 FORMAT(38X,80A1)
      GO TO 102
  103 IF(COL(7).EQ.BLANK) GO TO 800
      N=N+1
      OP=PACK(COL(7),COL(8),BLANK,BLANK)
      OPER=PACK(COL(7),COL(8),COL(9),COL(10))
```

```
107   DO 108 INSTR=1,20
      IF(OPCODE(INSTR).EQ.OP) GO TO 109
108   CONTINUE
      INSTR=21
109   IF(INSTR.LE.11) GO TO 800
      IF(INSTR.EQ.20.AND.COL(9).NE.BLANK) GO TO 20
1091  IF(ITOP(LST,LCURR(2))) 100,110,100
110   CALL NEWBOT(OPER,LST)
      IF(INSTR.EQ.18) GO TO 18
      IF(INSTR.GE.14.AND.INSTR.LE.19) CALL DISECT(COL,INSTR,LST)
      IF(INSTR.GT.13) GO TO 100
      I=0
      IF(INSTR.EQ.12) CALL BR(COL,LBR,I,LST)
      IF(INSTR.EQ.13) CALL TR(COL,LBR)
      M=NEWBOT(NAME(LBR),LST)
      CALL SETIND(-2,-1,-1,M)
      IF(I.NE.0) GO TO 100
800   NEXT=MAJOR (OPER,INSTR,LCURR,MINOR,NPR)
      IF(INSTR.EQ.100) GO TO 400
      IF(NEXT) 801,100,801
801   IF(INSTR.EQ.7) GO TO 806
      IF(INSTR.EQ.11) GO TO 808
      IF(ILIST.EQ.0) WRITE(6,802) NEXT,N,(COL(I),I=7,80)
802   FORMAT(I34,I8,2X,74A1)
      GO TO 102
806   IF(ILIST.EQ.0) WRITE(6,807) OPER,NEXT,N,(COL(I),I=7,80)
807   FORMAT(18X,A3,I13,I8,2X,74A1)
      GO TO 102
808   IF(ILIST.EQ.0) WRITE(6,809) OPER,NEXT,N,(COL(I),I=7,80)
809   FORMAT(24X,A3,I7,I8,2X,74A1)
      GO TO 102
18    CALL LR(COL,LST)
100   IF(MINOR) 804,805,804
804   NEXT=MINOR
      MINOR=0
      GO TO 801
805   IF(ILIST.EQ.0) WRITE(6,803) N,(COL(I),I=7,80)
803   FORMAT(I42,2X,74A1)
      GO TO 102
20    INSTR=COL(9)/16777216+15
      IF(INSTR.EQ.4.OR.INSTR.GT.7) GO TO 1091
      INSTR=INSTR+1
      IF(INSTR.GT.3) INSTR=INSTR+2
      IF(INSTR.EQ.6) INSTR=7
      IF(INSTR.EQ.3) CALL FN2(COL,INSTR,LST)
      GO TO 800
400   CONTINUE
      IF(ILIST.EQ.0) WRITE(6,99) TITLE
      IF(ITRACE.EQ.0) CALL TRACE(LCORSE,NLOOP)
      IF(NPER.EQ.0) GO TO 401
      WRITE(6,99) TITLE
```

```
      CALL STRACE(LCORSE,NLOOP,NPER,NVAR,IPUNCH)
401   IF(ICHART.NE.0) GO TO 402
      WRITE(6,99) TITLE
      CALL CHART(LCORSE)
402   DO 93 I=1,10
93    CALL MTLIST(LW(I))
      GO TO 91
      END
      SUBROUTINE MODEL(N,IX,IREF,ICELL,NODE,BRANCH)
C     MODEL 1 - RANDOM RESPONSE MODEL
      INTEGER IX(1),IR/5093/
      LOGICAL NODE,BRANCH
      IF(NODE) GO TO 99
      IX(1)=IX(1)+1
      RETURN
99    CONTINUE
      CALL RANDU(IR,IY,R)
      IR=IY
      IF(R.LT.0.5) BRANCH=.FALSE.
      RETURN
      END
      SUBROUTINE FN2(COL,INSTR,LST)
      INTEGER COL(1)
      RETURN
      END
      SUBROUTINE STRACE(LCORSE,NLOOP,NPER,NVAR,IPUNCH)
      COMMON LAVS,LW(10)
      LOGICAL IVAL,TOUT,NODE,BRANCH
      INTEGER*2 C(31),S(528)
      INTEGER RR(6),IUN2RR(6),IX(250)
      NPATH=0
      TOUT=.FALSE.
78    DO 85 I=1,31
85    C(I)=0
      DO 86 I=1,528
86    S(I)=0
      MEMORY=LIST(3)
      LST=LCORSE
      ICELL=LST
      LENGTH=0
      LASTEP=0
      NEXT=LST
      IF(NVAR.EQ.0) GO TO 99
      DO 98 I=1,NVAR
98    IX(I)=0
99    NRING=0
      LRING=0
100   ICELL=LRIN(ICELL)
      ID=IDIN(ICELL)
      IF(ID-1) 200,301,500
101   LENGTH=LENGTH+1
```

```
      NODE=.FALSE.
      CALL MODEL(NPATH,IX,LST,ICELL,NODE,BRANCH)
      IF(ID.NE.6.AND.ID.NE.7) GO TO 100
      DO 1011 I=6,ID
1011  ICELL=LRIN(ICELL)
      GO TO 100
200   IF(ID.EQ.0) GO TO 101
      ID=-ID
      IF(ID.EQ.2) GO TO 301
      IF(ID.LE.4) GO TO 250
      IF(ID-12) 102,12,202
102   IF(ID.NE.6.AND.ID.NE.7) GO TO 100
      GO TO 101
12    IF(IVAL(ICELL,C,S)) GO TO 101
      ICELL=LRIN(ICELL)
      GO TO 101
202   CALL EXCUTE(ICELL,ID,C,S,RR,IUN2,IUN2RR)
      GO TO 101
250   IF(ID.NE.4.OR.INHALT(LST).NE.6) GO TO 251
      IF(MADLST(INHALT(ICELL)).EQ.LST) GO TO 100
      IUN1=INHALT(LRIN(ICELL))
      IF(IUN1-IUN2) 301,100,301
251   IF(ID.EQ.4.AND.TOUT) GO TO 301
      BRANCH=.TRUE.
      NODE=.TRUE.
      CALL MODEL(NPATH,IX,LST,ICELL,NODE,BRANCH)
      IF(.NOT.BRANCH) GO TO 100
      IF(ID.EQ.3) TOUT=.TRUE.
301   LST=MADLST(INHALT(ICELL))
      IF(LST-11) 303,302,304
302   LST=LASTEP
      GO TO 304
303   LSTT=RR(LST-1)
      IF(IFIND(LSTT,LW(7),M)) 304,3031,304
3031  LST=LSTT
      IUN2=IUN2RR(LST-1)
304   NEXT=LLIN(NEXT)
      I=1
      IF(NRING.NE.0) GO TO 402
400   IF(ID.EQ.1) GO TO 403
      IF(IFIND(LST,MEMORY,NEXT)) 403,401,403
401   LRING=LST
402   IF(LRING.EQ.LST) NRING=NRING+1
      IF(NRING.EQ.NLOOP) GO TO 500
403   M=NEWTOP(LST,MEMORY)
      CALL SETIND(-ID,-1,-1,M)
      IF(I.EQ.2) GO TO 100
      IF(INHALT(LST).EQ.11) TOUT=.FALSE.
      ICELL=LST
      IF(INHALT(LST).NE.11) GO TO 405
      IF(LST.EQ.LASTEP) GO TO 404
```

```
      LASTEP=LST
      IUN2=0
  404 IUN2=IUN2+1
  405 IF(NRING.EQ.0.OR.INHALT(NEXT).EQ.LST) GO TO 100
      CALL IPOPUP(LST,MEMORY)
      NRING=0
      LRING=0
      I=2
      GO TO 400
  500 CONTINUE
      CALL PATH(MEMORY,LRING,NLOOP,NPATH,LENGTH,'STUDENT ')
      CALL IRALST(MEMORY)
      IF(NVAR.EQ.0) GO TO 600
      M=(NVAR+13)/14
      DO 540 J=1,M
      IA=J*14-13
      IB=J*14-J/M*(J*14-NVAR)
      WRITE(6,510) (I,I=IA,IB)
  510 FORMAT(1H05X'VARIABLE'14I6)
      WRITE(6,520) (IX(I),I=IA,IB)
  520 FORMAT(14X14I6)
      IF(IPUNCH.NE.0) WRITE(7,530) NPATH,J,(IX(I),I=IA,IB)
  530 FORMAT(16I5)
  540 CONTINUE
  600 IF(NPATH.NE.NPER) GO TO 78
      RETURN
      END
      SUBROUTINE TRACE(LCORSE,NLOOP)
      COMMON LAVS,LW(10)
      LOGICAL IVAL,TOUT
      INTEGER*2 C(31),S(528)
      INTEGER RR(6),IUN2RR(6)
      LRECRD=LIST(3)
      LISTEP=LIST(3)
      LSHUT=0
      NPATH=0
      TOUT=.FALSE.
   78 LDOOR=0
      DO 85 I=1,31
   85 C(I)=0
      DO 86 I=1,528
   86 S(I)=0
      MEMORY=LIST(3)
      LST=LCORSE
      ICELL=LST
      LENGTH=0
      LAST=0
      LASTEP=0
      ISHUT=0
      NEXT=LST
   99 NRING=0
```

```
      LRING=0
100   ICELL=LRIN(ICELL)
      ID=IDIN(ICELL)
      IF(ID-1) 200,301,500
101   LENGTH=LENGTH+1
      GO TO 100
200   IF(ID.EQ.0) GO TO 101
      ID=-ID
      IF(ID.EQ.2) GO TO 301
      IF(ID.LE.4) GO TO 250
      IF(ID-12) 102,12,202
102   IF(ID.NE.7) GO TO 100
      LENGTH=LENGTH-1
      GO TO 100
12    IF(IVAL(ICELL,C,S)) GO TO 101
      ICELL=LRIN(ICELL)
      GO TO 101
202   CALL EXCUTE(ICELL,ID,C,S,RR,IUN2,IUN2RR)
      GO TO 101
250   IF(ID.NE.4.OR.INHALT(LST).NE.6) GO TO 251
      IF(MADLST(INHALT(ICELL)).EQ.LST) GO TO 100
      IUN1=INHALT(LRIN(ICELL))
      IF(IUN1-IUN2) 301,100,301
251   IF(ID.EQ.4.AND.TOUT) GO TO 301
      IF(LST.NE.LSHUT) GO TO 290
      IF(LDOOR.NE.LAST) ISHUT=LSHUT
      LAST=LDOOR
      LREC=LLIN(LREC)
      LSHUT=INHALT(LREC)
      GO TO 100
290   IF(ID.NE.3) GO TO 295
      IF(IFIND(LST,LISTEP,M)) 294,100,294
294   TOUT=.TRUE.
      GO TO 296
295   IF(NRING.EQ.0) GO TO 296
      IF(MADLST(INHALT(ICELL)).NE.INHALT(LLIN(NEXT))) GO TO 100
296   LDOOR=LST
301   LST=MADLST(INHALT(ICELL))
      IF(LST-11) 303,302,304
302   LST=LASTEP
      GO TO 304
303   LSTT=RR(LST-1)
      IF(IFIND(LSTT,LW(7),M)) 304,3031,304
3031  LST=LSTT
      IUN2=IUN2RR(LST-1)
304   NEXT=LLIN(NEXT)
      I=1
      IF(NRING.NE.0) GO TO 402
400   IF(IC.EQ.1) GO TO 403
      IF(IFIND(LST,MEMORY,NEXT)) 403,401,403
401   LRING=LST
```

```
402   IF(LRING.EQ.LST) NRING=NRING+1
      IF(NRING.EQ.NLOOP) GO TO 500
403   M=NEWTOP(LST,MEMORY)
      CALL SETIND(-ID,-1,-1,M)
      IF(I.EQ.2) GO TO 100
      IF(INHALT(LST).EQ.11) TOUT=.FALSE.
      ICELL=LST
      IF(INHALT(LST).NE.11) GO TO 405
      IF(LST.EQ.LASTEP) GO TO 404
      LASTEP=LST
      IUN2=0
404   IUN2=IUN2+1
405   IF(NRING.EQ.0.OR.INHALT(NEXT).EQ.LST) GO TO 100
      CALL IPOPUP(LST,MEMORY)
      NRING=0
      LRING=0
      I=2
      GO TO 400
500   CONTINUE
      CALL PATH(MEMORY,LRING,NLOOP,NPATH,LENGTH,'PATH NO.')
      CALL IRALST(MEMORY)
      IF(LDOOR.EQ.0) GO TO 600
      IF(INHALT(LDOOR).EQ.11) GO TO 501
      M=NEWTOP(LDOOR,LRECRD)
5002  IF(LDOOR.NE.LAST) GO TO 5004
5003  CALL IPOPUP(LSHUT,M)
      IF(LSHUT.NE.ISHUT) GO TO 5003
5004  LREC=LLIN(MADLST(LRECRD))
      LSHUT=INHALT(LREC)
      GO TO 78
501   CALL NEWTOP(LDOOR,LISTEP)
      M=LRECRD
      GO TO 5002
600   CALL IRALST(LRECRD)
      CALL IRALST(LISTEP)
      RETURN
      END
      SUBROUTINE PATH(MEMORY,LRING,NLOOP,NPATH,LENGTH,TITLE)
      INTEGER CHANEL(32),IP(8),BLANK/' '/,TITLE(2)
      INTEGER TOU/'T-OU'/,T/'T'/,MIS/'MIS-'/,MA/'M'/,LOOP/'LOOP'/
      NPATH=NPATH+1
      WRITE(6,99) TITLE,NPATH,LENGTH
 99   FORMAT(1HO,2A4,I5,10XI5,' STATEMENTS EXECUTED')
      IF(LRING.EQ.0) GO TO 110
      NRING=1
 100  ID=IDIN(LRIN(MADLST(MEMORY)))
      CALL IPOPUP(LST,MEMORY)
      IF(LST.NE.LRING) GO TO 100
      NRING=NRING+1
      IF(NRING.NE.NLOOP) GO TO 100
      M=NEWTOP(LST,MEMORY)
```

```
      CALL SETIND(ID,-1,-1,M)
110   ICELL=MADLST(MEMORY)
      N=0
111   ICELL=LLIN(ICELL)
112   ID=IDIN(ICELL)
      IF(ID.GE.2) GO TO 700
      IF(ID.EQ.-3) GO TO 300
      IF(ID.EQ.-4) GO TO 400
1120  IREF=INHALT(ICELL)
      INSTR=INHALT(IREF)
      IF(INSTR.LT.0) GO TO 203
      IF((N+3).GT.32) GO TO 500
113   N=N+2
      CALL HOLLER(CHANEL(N),IREF,IP)
      M=2
      DO 201 I=1,5
      IF(IP(6-I).EQ.BLANK) GO TO 200
      IP(6-I+M)=IP(6-I)
      GO TO 201
200   M=M+1
201   CONTINUE
      DO 202 I=1,M
202   IP(I)=BLANK
      CHANEL(N-1)=IPACK(IP(1),IP(2),IP(3),IP(4))
      CHANEL(N)=IPACK(IP(5),IP(6),IP(7),IP(8))
      M=1
      GO TO 2041
203   M=(3-INSTR)/4
      IF((N+M+1).GT.32) GO TO 600
204   N=N+1
      CHANEL(N)=BLANK
2041  DO 205 I=1,M
      IREF=LRIN(IREF)
205   CHANEL(N+I)=INHALT(IREF)
      N=N+M
206   IF(N.NE.32) GO TO 111
      ASSIGN 111 TO IBACK
207   WRITE(6,208) (CHANEL(I),I=1,N)
208   FORMAT(2X32A4)
      N=0
      GO TO IBACK, (111,301,401,113,204,701,1120,7001)
300   IF((N+2).GT.32) GO TO 302
301   N=N+2
      CHANEL(N-1)=BLANK
      IF(N.EQ.2) N=3
      CHANEL(N-1)=TOU
      CHANEL(N)=T
      GO TO 450
302   ASSIGN 301 TO IBACK
      GO TO 207
400   IF((N+2).GT.32) GO TO 402
```

```
401   N=N+2
      CHANEL(N-1)=BLANK
      IF(N.EQ.2) N=3
      CHANEL(N-1)=MIS
      CHANEL(N)=MA
450   IF(N.NE.32) GO TO 1120
      ASSIGN 1120 TO IBACK
      GO TO 207
402   ASSIGN 401 TO IBACK
      GO TO 207
500   ASSIGN 113 TO IBACK
      GO TO 207
600   ASSIGN 204 TO IBACK
      GO TO 207
700   ASSIGN 701 TO IBACK
      IF(LRING.EQ.0) GO TO 7003
      IF((N+2).GT.32) GO TO 7002
7001  N=N+2
      CHANEL(N-1)=BLANK
      CHANEL(N)=LOOP
      ASSIGN 701 TO IBACK
      GO TO 207
7002  ASSIGN 7001 TO IBACK
      GO TO 207
7003  IF(N.NE.0)  GO TO 207
701   RETURN
      END
      FUNCTION ISWICH(ICELL,IN)
      ISWICH=IUNPAK(IN,4)/16777216+15
      IF(IDIN(ICELL).NE.-6) GO TO 5
      ICELL=LRIN(ICELL)
      IALPHA=IUNPAK(INHALT(ICELL),1)/16777216+63
      J=10
      GO TO 8
5     IALPHA=ISWICH+48
      IF(ISWICH.LT.0) GO TO 7
      IALPHA=-31
      K=1
      J=10
      IF(ISWICH.LE.9) GO TO 9
7     ISWICH=0
      J=1
8     IF(IALPHA.GT.10) IALPHA=IALPHA-7
      IF(IALPHA.GT.0) K=16
9     ISWICH=K*(ISWICH+J*(IUNPAK(IN,3)/16777216+15))+IALPHA+32
      RETURN
      END
      FUNCTION ICOUNT(ICELL,IN)
      ICOUNT=IUNPAK(IN,4)/16777216+15
      J=10
      IF(ICOUNT.LE.9) GO TO 13
```

```
      ICOUNT=0
      J=1
   13 ICOUNT=ICOUNT+J*(IUNPAK(IN,3)/16777216+15)+1
      RETURN
      END
      SUBROUTINE EXCUTE(ICELL,INSTR,C,S,RR,IUN2,IUN2RR)
      INTEGER*2 C(1),S(1)
      INTEGER RR(1),IUN2RR(1),IP(2),CEE/'C'/
      K=1
      IS=0
      ICELL=LRIN(ICELL)
      IF(INSTR.NE.18) GO TO 12
      LABEL=INHALT(ICELL)
      ICELL=LRIN(ICELL)
      N=-4-IDIN(ICELL)
      DO 11 I=1,N
   11 ICELL=LRIN(ICELL)
      I=IUNPAK(INHALT(ICELL),4)/16777216+16
      RR(I)=LABEL
      IUN2RR(I)=IUN2
      RETURN
   12 IP(K)=INHALT(ICELL)
      IF(IDIN(ICELL).EQ.-5) GO TO 14
      IC=ICOUNT(ICELL,IP(K))
      IP(K)=C(IC)
      IF(K-1) 14,14,15
   14 ICELL=LRIN(ICELL)
      K=2
      IN=INHALT(ICELL)
      IF(IUNPAK(IN,2).EQ.CEE) GO TO 12
      IS=ISWICH(ICELL,IN)
      IP(2)=S(IS)
   15 INST=INSTR-13
      GO TO (1,2,3,4,4,6),INST
    1 IN=IP(2)+IP(1)
      GO TO 16
    2 IN=IP(2)-IP(1)
      GO TO 16
    3 IN=IP(2)*IP(1)
      GO TO 16
    4 IN=IP(2)/IP(1)
      GO TO 16
    6 IN=IP(1)
   16 IF(IS.GT.0) GO TO 17
      C(IC)=IN
      RETURN
   17 S(IS)=IN
      RETURN
      END
      LOGICAL FUNCTION IVAL(ICELL,C,S)
      INTEGER*2 C(1),S(1)
```

```
      INTEGER CEE/'C'/,REL(6)/' L',' LE',' E',' NE',' GE',' G'/
      ICELL=LRIN(ICELL)
      IN=INHALT(ICELL)
      IF(IUNPAK(IN,2).EQ.CEE) GO TO 10
      I1=ISWICH(ICELL,IN)
      I1=S(I1)
      I=3
      GO TO 12
10    I1=ICOUNT(ICELL,IN)
      I1=C(I1)
      ICELL=LRIN(ICELL)
      IN=INHALT(ICELL)
      DO 11 I=1,6
      IF(REL(I).EQ.IN) GO TO 12
11    CONTINUE
12    ICELL=LRIN(ICELL)
      I2=INHALT(ICELL)
      IF(IDIN(ICELL).EQ.-5) GO TO 13
      I2=ICOUNT(ICELL,I2)
      I2=C(I2)
13    GO TO (1,2,3,4,5,6),I
1     IVAL=I1.LT.I2
      RETURN
2     IVAL=I1.LE.I2
      RETURN
3     IVAL=I1.EQ.I2
      RETURN
4     IVAL=I1.NE.I2
      RETURN
5     IVAL=I1.GE.I2
      RETURN
6     IVAL=I1.GT.I2
      RETURN
      END
      FUNCTION MAJOR (OPER,INSTR,LCURR,MINOR,NPR)
      COMMON LAVS,LW(10)
      INTEGER LCURR(1)
      INTEGER   CONTIN(11)/6,6,7,8,9,9,6,9,8,7,7/
      INTEGER   BRANCH(11)/0,10,3,4,5,1,0,5,4,3,2/
      INTEGER   DECIDE(11,11)/1,5*0,1,4*3,2,6*0,4*3,2,6*0,2*3,0,3,2,6*0,
     13,0,2*3,2,7*0,3*3,2,6*0,3*3,0,2,4,5*0,3*3,0,6*2,0,3*3,0,7*5,3*3,0,
     210*0,3,11*0/
      INST=MINO(INSTR,11)
      LISTEX=0
      MAJOR=0
      DO 710 I=1,2
      K=LCURR(I)
700   IF(IPOPUP(LST,K)) 7091,701,7091
701   CALL IPOPUP(LAST,K)
      ICOND=DECIDE(INST,LAST)
      IF(I.EQ.1) GO TO 704
```

```
      ILST=LST
7011  IF(IPOPUP(NLST,K)) 704,702,704
702   CALL IPOPUP(NLAST,K)
      IF(DECIDE(INST,NLAST).NE.ICOND) GO TO 703
      IF(NLAST.NE.LAST.AND.ICOND.EQ.3) GO TO 703
      CALL NEWBOT(NAME(ILST),NLST)
      ILST=NLST
      GO TO 7011
703   NLST=LCOPY(ILST,NLST)
      CALL NEWTOP(NLAST,NEWTOP(NLST,K))
704   GO TO (1,2,3,4,5),ICOND
      IF(INSTR.GT.11) GO TO 711
      IF(LISTEX.NE.0) GO TO 707
      IF(INSTR.NE.7) GO TO 705
7041  CALL IPOPUP(LISTEX,LW(8))
      CALL ITOP(LISTEX,LW(8))
      CALL NEWBOT(MADLST(LIST(2)),LW(8))
      NPR=NPR+1
      CALL STRIND(NPR,LISTEX)
      GO TO 706
705   LISTEX=MADLST(LIST(2))
      CALL STRIND(INSTR,LISTEX)
706   CALL NEWBOT(OPER,LISTEX)
      IF(INSTR.NE.6) GO TO 707
      IUN1=IUN1+1
      M=NEWBOT(IUN1,LISTEX)
      CALL SETIND(-8,-1,-1,M)
707   NEXT=LISTEX
      IF(LST) 710,710,708
1     NEXT=LASTUN
      GO TO 708
2     CALL ITOP(NEXT,LRIN(MADLST(LW(8))))
      GO TO 708
3     GO TO (31,32),I
31    K=NEWTOP(LAST,NEWTOP(LST,K))
      GO TO 700
32    NEXT=MADLST(LIST(2))
      MINOR=NEXT
      CALL STRIND(0,NEXT)
      K=NEWTOP(LAST,NEWTOP(NEXT,K))
      GO TO 709
4     CALL NEWBOT(LST,NEWBOT(10,LCURR(1)))
      GO TO 700
5     NEXT=LW(9)
708   IF(I.EQ.2.OR.LAST.EQ.10) GO TO 709
      M=NEWTOP(NAME(NEXT),LRIN(LST))
      CALL SETIND(-4,-1,-1,M)
      IF(LAST.EQ.2) CALL SETIND(-3,-1,-1,M)
      IF(ICOND.EQ.2) CALL SETIND(-2,-1,-1,M)
      GO TO 700
709   M=NEWBOT(NAME(NEXT),LST)
```

```
      IF(LAST.EQ.10.OR.NEXT.EQ.LW(9).OR.ICOND.EQ.2)
     1 CALL SETIND(-2,-1,-1,M)
      GO TO 700
7091  IF(LISTEX.EQ.0.AND.INSTR.EQ.7.AND.I.EQ.2) GO TO 7041
710   CONTINUE
      IF(LISTEX.EQ.0) RETURN
      LAST=CONTIN(INSTR)
      CALL NEWBOT(LISTEX,NEWBOT(LAST,LCURR(2)))
      IF(INSTR.NE.11) GO TO 7102
      LW(9)=MADLST(LW(10))
      IF(LISTMT(LCURR(1))) 7102,7101,7102
7101  LW(9)=LISTEX
7102  IF(LAST.NE.6) CALL NEWBOT(LISTEX,NEWBOT(BRANCH(INSTR),LCURR(1)))
      IF(INSTR.EQ.6) LASTUN=LISTEX
      IF(INSTR.EQ.7) LW(9)=MADLST(LW(10))
      IF(INSTR.EQ.11) IUN1=0
711   MAJOR=LISTEX
      RETURN
      END
      SUBROUTINE CHART(LCORSE)
      COMMON LAVS,LW(10)
      INTEGER COND(16),CELL(16),CHANEL(32),BLANK/' '/
      INTEGER IP(8),BAR/' | '/,MINUS/'-'/
      NC=16
      CALL MTLIST(LW(8))
      LST=LRIN(LCORSE)
      JUNK=LIST(3)
89    DO 90 I=1,NC
90    COND(I)=6
      COND(9)=8
      CELL(9)=LST
      IBEGIN=0
91    IEND=0
      DO 92 I=1,NC
      IF(COND(I).NE.6) IEND=1
      CHANEL(2*I-1)=BLANK
92    CHANEL(2*I)=BLANK
      IF(IEND) 93,19,93
93    DO 6 I=1,NC
      ICOND=COND(I)
      ICELL=CELL(I)
      GO TO (1,2,3,4,4,6,4,8,9),ICOND
1     ID=IDIN(ICELL)
      IF(ID-1) 10,11,12
10    IF(ID.EQ.0) GO TO 1002
      N=-4-ID
      IF(N-1) 1001,1002,1003
1001  ICOND=2
      GO TO 2
1002  N=1
1003  IF(ID.EQ.-8) GO TO 128
```

```
      IF(N.GT.3) N=1
      DO 1004 J=1,N
      CHANEL(2*I-2+J)=INHALT(ICELL)
      IF(COND(I).EQ.2.OR.COND(I).EQ.4) GO TO 1004
      ID=IDIN(LRIN(ICELL))
      IF(ID.LE.-2.AND.ID.GE.-4) GO TO 13
1004  ICELL=LRIN(ICELL)
      IF(COND(I).NE.4.OR.IDIN(ICELL).NE.1) GO TO 1005
      LL=LLIN(CELL(I))
      IF(IDIN(LL).GE.2.AND.INHALT(LL).LT.0) COND(I)=9
1005  CELL(I)=ICELL
101   IF(IDIN(CELL(I)).GE.2) COND(I)=4
      GO TO 6
102   IN=INHALT(ICELL)
      IPO=BLANK
      IF(IN.GE.0) GO TO 1021
      IPO=MINUS
      IN=-IN
1021  CALL HOLLER(CHANEL(2*I-1),IN,IP)
      CHANEL(2*I-1)=IPACK(IPO,IP(1),IP(2),IP(3))
      CHANEL(2*I)=IPACK(IP(4),IP(5),IP(6),IP(7))
      IF(COND(I).EQ.2.OR.COND(I).EQ.4) GO TO 1022
      ID=IDIN(LRIN(ICELL))
      IF(ID.LE.-2.AND.ID.GE.-4) GO TO 13
1022  CELL(I)=LRIN(ICELL)
      GO TO 101
11    CHANEL(2*I-1)=BAR
      CELL(I)=MADLST(INHALT(ICELL))
      GO TO 6
12    INSTR=INHALT(ICELL)
      IF(COND(I).EQ.4) GO TO 125
120   IF(INSTR.LT.100.OR.IBEGIN.EQ.0) GO TO 121
      CALL PR (ICELL,INSTR)
125   COND(I)=3
      IF(INSTR.LT.0) COND(I)=4
      GO TO 127
121   IF(IDIN(ICELL)-3) 122,122,123
122   IF(INSTR) 126,127,127
123   IF(IFIND(ICELL,JUNK,N)) 126,125,126
126   CALL NEWTOP(ICELL,JUNK)
127   IF(INSTR.LT.0) GO TO 128
      CALL HOLLER(CHANEL(2*I-1),ICELL,IP)
      CELL(I)=LRIN(ICELL)
      IBEGIN=1
      GO TO 6
128   ICELL=LRIN(ICELL)
      CELL(I)=ICELL
      GO TO 1
13    ICOND=13
      IF(N.GT.2) GO TO 22
2     N=NUCHAN(COND,I,NC)
```

```
      IF(N.NE.I) GO TO 25
      CHANEL(2*I-1)=BAR
22    COND(I)=2
      IF(ICOND.EQ.2) GO TO 6
      ICELL=CELL(I)
      GO TO 1
25    CALL ARTIST(ICELL,ICOND,CHANEL,I,N)
      CELL(I)=LRIN(ICELL)
      CELL(N)=MADLST(INHALT(ICELL))
      COND(I)=1
      COND(N)=8
      IF(N.GT.I) COND(N)=7
      GO TO 101
3     COND(I)=4
      GO TO 1
4     COND(I)=ICOND+1
      GO TO 6
9     COND(I)=3
      GO TO 11
8     COND(I)=1
      CHANEL(2*I-1)=BAR
6     CONTINUE
      WRITE(6,31) CHANEL
31    FORMAT(1X,16(2A4))
      GO TO 91
19    CONTINUE
      DO 193 I=1,2
191   IF(IPOPUP(LST,LW(9-I))) 193,192,193
192   IF(IFIND(LST,JUNK,N)) 89,191,89
193   CONTINUE
      CALL IRALST(JUNK)
      RETURN
      END
      SUBROUTINE HOLLER(IHOLL,ICELL,IP)
C     CHANGES 5 PLACE INTEGER TO HOLLERITH
      INTEGER IHOLL(1),IP(8),BLANK/' '/,ZERO/'0'/
      IR=ICELL
      J=100000
      N=0
      DO 3 I=1,5
      J=J/10
      K=IR/J
      IF(K)1,1,2
1     IF(N) 3,3,2
2     N=N+1
      IP(N)=(K-16)*16777216
3     IR=IR-IR/J*J
      NP1=N+1
      DO 4 I=NP1,8
4     IP(I)=BLANK
      IF(N.EQ.0) IP(1)=ZERO
```

```
      IHOLL(1)=IPACK(IP(1),IP(2),IP(3),IP(4))
      IHOLL(2)=IPACK(IP(5),IP(6),IP(7),IP(8))
      RETURN
      END
      SUBROUTINE PR(ICELL,INSTR)
      COMMON LAVS,LW(10)
      L=MADLST(LW(8))
    1 L=LRIN(L)
      IF(IDIN(L)-2) 2,3,3
    2 K=INHALT(L)
      IF(INSTR-INHALT(K)) 3,4,1
    3 CALL NEWBOT(ICELL,L)
    4 RETURN
      END
      SUBROUTINE ARTIST(ICELL,ICOND,CHANEL,I,N)
    C DRAWS LINE FROM CELL AT CHANNEL I TO CHANNEL N.
      INTEGER IP(4),CHANEL(1),STROKE/'____'/,BLANK/'    '/,I082/'  '/
      INTEGER ILEFT/'_|  '/,IRIGHT/'  |__'/,NLEFT/'  __'/,NRIGHT/'_    '/
      INTEGER TOU/'T-OU'/,T/'T'/,MIS/'MIS-'/,M/'M'/
      INTEGER TS/'T___'/,MS/'M___'/
      I2M1=2*I-1
      N2M1=2*N-1
      IF(N.GT.I) GO TO 2
      I2M2=2*I-2
      DO 1 J=N2M1,I2M2
    1 CHANEL(J)=STROKE
      CHANEL(N2M1)=NLEFT
      IF(ICOND.NE.2) GO TO 4
      CHANEL(I2M1)=ILEFT
      GO TO 5
    2 I2=2*I
      I2P1=2*I+1
      DO 3 J=I2P1,N2M1
    3 CHANEL(J)=STROKE
      CHANEL(N2M1)=NRIGHT
      IF(ICOND.NE.2) GO TO 30
      CHANEL(I2M1)=IRIGHT
      CHANEL(I2)=STROKE
      GO TO 5
   30 DO 31 J=1,4
      IP(J)=IUNPAK(CHANEL(I2),J)
      IF(IP(J).EQ.BLANK.OR.IP(J).EQ.I082) IP(J)=STROKE
   31 CONTINUE
      CHANEL(I2)=IPACK(IP(1),IP(2),IP(3),IP(4))
      DO 32 J=3,4
      IP(J)=IUNPAK(CHANEL(I2M1),J)
      IF(IP(J).EQ.BLANK.OR.IP(J).EQ.I082) IP(J)=STROKE
   32 CONTINUE
      CHANEL(I2M1)=IPACK(IUNPAK(CHANEL(I2M1),1),IUNPAK(CHANEL(I2M1),2),
     1 IP(3),IP(4))
    4 ICELL=LRIN(ICELL)
```

```
5     ID=IDIN(ICELL)
      IF(ID+3) 7,6,8
6     CHANEL(N2M1)=TOU
      CHANEL(2*N)=T
      IF(N.LT.I) CHANEL(2*N)=TS
      RETURN
7     CHANEL(N2M1)=MIS
      CHANEL(2*N)=M
      IF(N.LT.I) CHANEL(2*N)=MS
8     RETURN
      END
      SUBROUTINE LR(COL,LST)
      INTEGER COL(1),IP(11),BLANK/' '/
      CALL SETIND(-18,-1,-1,LLIN(LST))
      CALL LIMTER(COL(11),61,' ',1,N)
      CALL LIMTER(COL(11),N,'¬/',2,K)
      NEXT=NULABL(COL(11),K)
      N=NEWBOT(NEXT,LST)
      CALL SETIND(-8,-1,-1,N)
      N=MINO(3,K/4+1)
      DO 1 I=1,3
      IP(I)=BLANK
      IF(I.LE.K) IP(I)=COL(I+10)
1     CONTINUE
      IN=IPACK(BLANK,IP(1),IP(2),IP(3))
      M=NEWBOT(IN,LST)
      IF(K.GT.3) CALL SETIND(-N-4,-1,-1,M)
      IF(K.LE.3) GO TO 4
      DO 2 I=4,7
      IP(I)=BLANK
      IF(I.LE.K) IP(I)=COL(I+10)
2     CONTINUE
      IN=IPACK(IP(4),IP(5),IP(6),IP(7))
      CALL NEWBOT(IN,LST)
      IF(K.LE.7) GO TO 4
      DO 3 I=8,11
      IP(I)=BLANK
      IF(I.LE.K) IP(I)=COL(I+10)
3     CONTINUE
      IN=IPACK(IP(8),IP(9),IP(10),IP(11))
      CALL NEWBOT(IN,LST)
4     CALL NEWBOT(PACK(BLANK,COL(K+13),COL(K+14),COL(K+15)),LST)
      RETURN
      END
      FUNCTION LCOPY(ILST,NLST)
      LCOPY=NLST
      K=ILST
1     K=LRIN(K)
      IF(IDIN(K)-1) 2,3,4
2     M=NEWBOT(INHALT(K),LCOPY)
      CALL SETIND(IDIN(K),-1,-1,M)
```

```
      GO TO 1
    3 K=MADLST(INHALT(K))
      GO TO 1
    4 RETURN
      END
      FUNCTION NULABL(LABEL,N)
C     RETURNS NAME OF LABEL LIST AND STORES LABEL AND LIST ON LW(7) IF
C     NOT ALREADY THERE
      COMMON LAVS,LW(10)
      INTEGER LABEL(1),LAB(3)
      CALL SPACK(LABEL,N,LAB,M)
      K=MADLST(LW(7))
    2 K=LLIN(K)
      IF(IDIN(K)-2) 3,6,6
    3 L=INHALT(K)
      IF(INHALT(L)+N) 2,4,2
    4 DO 5 I=1,M
      L=LRIN(L)
      IF(INHALT(L).NE.LAB(I)) GO TO 2
    5 CONTINUE
      NULABL=INHALT(K)
      RETURN
    6 NULABL=MADLST(LIST(2))
      CALL NEWBOT(NULABL,LW(7))
      CALL STRIND(-N,NULABL)
      DO 7 I=1,M
    7 LAB(I)=NEWBOT(LAB(I),NULABL)
      IF(M.GE.2) CALL SETIND(-M-4,-1,-1,LAB(1))
      RETURN
      END
      FUNCTION NUCHAN(COND,I,NC)
C     SELECTS BRANCH CHANNEL
C     ICF IS THE CENTERING FACTOR
      INTEGER COND(1)
      ICF=3
      IM1=I-1
      DO 21 K=1,IM1
      J=I-K
      IF(COND(J)-6) 22,21,22
   21 CONTINUE
      J=0
   22 IP1=I+1
      DO 23 K=IP1,NC
      IF(COND(K)-6) 24,23,24
   23 CONTINUE
      K=NC+1
   24 L=I-(I-J+ICF-2)/ICF
      M=I+(K-I+ICF-2)/ICF
      NUCHAN=L
      IF(K-I.GT.I-J) NUCHAN=M
      RETURN
```

```
      END
      SUBROUTINE BR(COL,LBR,IFLAG,LST)
C     SETS UP BRANCH TO LABEL, LASTEP, PROBLEM, OR RETURN REGISTER.
      COMMON LAVS,LW(10)
      INTEGER COL(1),E/'E'/,P/'P'/,R/'R'/
      INTEGER INTGER(10)/'0','1','2','3','4','5','6','7','8','9'/
      IFLAG=1
      K=MINO(LIMTER(COL(11),61,' ',1,K),LIMTER(COL(11),61,' ',1,K))
      IF(LIMTER(COL(11),K,'¬/',2,N).EQ.K) IFLAG=0
      IF(IFLAG.EQ.1) CALL BRC(COL,LST,K,N)
      IF(N.GT.3) GO TO 302
      IF(COL(11).EQ.P) GO TO 304
      IF(COL(11).EQ.R) GO TO 304
C     LABEL
  302 LBR=NULABL(COL(11),N)
  303 RETURN
  304 IF(COL(12).EQ.R) GO TO 305
      IF(COL(11).EQ.P) GO TO 302
      IF(COL(12).NE.E) GO TO 302
      IF(N.NE.2) GO TO 302
C     LASTEP
      LBR=LW(9)
      RETURN
  305 DO 306 I=1,8
      IF(COL(13).EQ.INTGER(I)) GO TO 307
  306 CONTINUE
      GO TO 302
  307 IF(COL(11).EQ.R) GO TO 308
C     PR
      LBR=INHALT(MADNTP(LW(8),I))
      RETURN
  308 IF(I.GT.6) GO TO 302
C     RR
      LBR=MADLST(LW(I))
      IF(IDIN(LRIN(MADLST(LW(I))))) 309,303,309
  309 CALL NEWTOP(PACK('R','R',COL(13),' '),LW(I))
      RETURN
      END
      SUBROUTINE BRC(COL,LST,K,N)
      INTEGER COL(1),C/'C'/,BLANK/' '/,MINUS/'-'/
      CALL SETIND(-12,-1,-1,LLIN(LST))
      CALL LIMTER(COL(N+13),K-N-2,'¬/',2,M)
      IF(M.GE.3) GO TO 2
    1 IN=IPACK(BLANK,COL(N+13),COL(N+14),BLANK)
      GO TO 4
    2 IN=IPACK(BLANK,COL(N+13),COL(N+14),COL(N+15))
      IF(M.EQ.3) GO TO 4
      J=NEWBOT(IN,LST)
      CALL SETIND(-6,-1,-1,J)
      IN=IPACK(COL(N+16),BLANK,BLANK,BLANK)
    4 CALL NEWBOT(IN,LST)
```

```
      M=N+M+15
      IF(COL(N+13).NE.C) GO TO 5
      CALL LIMTER(COL(M),4,'¬/',2,L)
      IP3=BLANK
      IF(L.EQ.2) IP3=COL(M+1)
      CALL NEWBOT(PACK(' ',COL(M),IP3,' '),LST)
      M=M+L+2
5     NN=K-M+11
      IF(COL(M).NE.C) GO TO 6
      IP3=BLANK
      IF(NN.EQ.3) IP3=COL(M+2)
      CALL NEWBOT(PACK(' ',C,COL(M+1),IP3),LST)
      RETURN
6     J=NN
      IF(COL(M).EQ.MINUS) J=NN-1
      IN=0
      L=1
      DO 7 I=1,J
      IN=IN+(COL(11+K-I)/16777216+15)*L
7     L=L*10
      IF(J.NE.NN) IN=-IN
      M=NEWBOT(IN,LST)
      CALL SETIND(-5,-1,-1,M)
      RETURN
      END
      SUBROUTINE TR(CCL,NEXT)
C     CREATES TRANSFER TO LABEL OF NEW SEGMENT
      INTEGER COL(1),LABEL(3),BLANK/' '/,IP(4)
      N=MINO(LIMTER(COL(11),61,' ',1,N),LIMTER(COL(11),61,' ',1,N))
      CALL LIMTER(COL(11),N,'¬/',2,K)
      NEXT=MADLST(LIST(2))
      CALL STRIND(2+K-N,NEXT)
      CALL SPACK(COL(13+K),N-K-2,LABEL,N)
      DO 1 I=1,N
1     LABEL(I)=NEWBOT(LABEL(I),NEXT)
      IF(N.GE.2) CALL SETIND(-N-4,-1,-1,LABEL(1))
      CALL NEWBOT('SEG-',NEXT)
      DO 2 I=1,4
      IP(I)=COL(I+10)
      IF(I.GT.K) IP(I)=BLANK
2     CONTINUE
      CALL NEWBOT(PACK(IP(1),IP(2),IP(3),IP(4)),NEXT)
      RETURN
      END
      SUBROUTINE DISECT(COL,INSTR,LST)
      INTEGER COL(1),S/'S'/,C/'C'/,BLANK/' '/,IP(4),MINUS/'-'/
      N=MINO(LIMTER(COL(11),61,' ',1,N),LIMTER(COL(11),61,' ',1,N))
      IF(N.GT.11) RETURN
      CALL LIMTER(COL(11),N,'¬/',2,K)
      IF(K.EQ.N) RETURN
      IF(CCL(K+13).NE.S.AND.COL(K+13).NE.C) RETURN
```

```
      CALL SETIND(-INSTR,-1,-1,LLIN(LST))
      IF(COL(11).NE.C) GO TO 10
      IP4=BLANK
      IF(K.EQ.3) IP4=COL(13)
      CALL NEWBOT(PACK(' ',C,COL(12),IP4),LST)
      GO TO 12
10    J=K
      IF(COL(11).EQ.MINUS) J=K-1
      IN=0
      M=1
      DO 11 I=1,J
      IN=IN+(COL(K+11-I)/16777216+15)*M
11    M=M*10
      IF(J.NE.K) IN=-IN
      M=NEWBOT(IN,LST)
      CALL SETIND(-5,-1,-1,M)
12    J=N-K-2
      IF(J.GE.3) GO TO 3
      IN=IPACK(BLANK,COL(K+13),COL(K+14),BLANK)
      GO TO 5
3     IN=IPACK(BLANK,COL(K+13),COL(K+14),COL(K+15))
      IF(J.EQ.3) GO TO 5
      M=NEWBOT(IN,LST)
      CALL SETIND(-6,-1,-1,M)
      IN=IPACK(COL(K+16),BLANK,BLANK,BLANK)
5     CALL NEWBOT(IN,LST)
      RETURN
      END
      FUNCTION IFIND(M,L,K)
      K=MADLST(L)
1     K=LRIN(K)
      IF(IDIN(K)-2) 2,4,4
2     IF(INHALT(K)-M) 1,3,1
3     IFIND=0
      RETURN
4     IFIND=-1
      RETURN
      END
      FUNCTION LIMTER(STRING,LENGTH,DELIMT,M,N)
C     FINDS NUMBER OF CHARACTERS N UP TO DELIMT FOR STRING OF GIVEN
C     LENGTH. DELIMITER LENGTH M MUST NOT EXCEED 4 CHARACTERS (1 WORD).
      REAL STRING(1),DEL(4)
      DO 1 J=1,M
1     DEL(J)=UNPACK(DELIMT,J)
      N=LENGTH+1-M
      DO 3 I=1,N
      DO 2 J=1,M
      IF(STRING(I+J-1).NE.DEL(J)) GO TO 3
2     CONTINUE
      GO TO 4
3     CONTINUE
```

```
      I=LENGTH+1
    4 LIMTER=I-1
      N=LIMTER
      RETURN
      END
      SUBROUTINE SPACK(STRING,LENGTH,PACKED,N)
C     PACKS STRING OF GIVEN LENGTH FROM A1 INTO PACKED STRING OF LENGTH
C     N WORDS OF TYPE A4. BLANKS FILL ANY REMAINING SPACE.
      REAL STRING(1),BLANK/' '/,PACKED(1),EXTRA(4)
      N=(LENGTH+3)/4
      M=N-1
      IF(M)6,3,1
    1 DO 2 I=1,M
    2 PACKED(I)=PACK(STRING(4*I-3),STRING(4*I-2),STRING(4*I-1),
     1 STRING(4*I))
    3 NEXTRA=LENGTH-M*4
      DO 5 I=1,4
      EXTRA(I)=BLANK
      IF(I-NEXTRA)4,4,5
    4 EXTRA(I)=STRING(4*M+I)
    5 CONTINUE
      PACKED(N)=PACK(EXTRA(1),EXTRA(2),EXTRA(3),EXTRA(4))
    6 RETURN
      END
      FUNCTION PACK(I1,I2,I3,I4)
C     EXAMPLES-  IPACK('A','B','C','D') IS 'ABCD'
C                IPACK('ABCD','EFGH','IJKL','MNOP') IS 'AEIM'   ETC.
C     PROGRAMMED OCT/68 BY D. FLATHMAN
      ENTRY IPACK(I1,I2,I3,I4)
      K=I1
      IF(K.LT.0) K=-(K+1)
      IPACK=K/16777216*16777216
      K=I2
      IF(K.LT.0) K=-(K+1)
      K=K/16777216
      IF(ISIGN(1,I1)+ISIGN(1,I2).EQ.0) K=255-K
      IPACK=IPACK+K*65536
      K=I3
      IF(K.LT.0) K=-(K+1)
      K=K/16777216
      IF(ISIGN(1,I1)+ISIGN(1,I3).EQ.0) K=255-K
      IPACK=IPACK+K*256
      K=I4
      IF(K.LT.0) K=-(K+1)
      K=K/16777216
      IF(ISIGN(1,I1)+ISIGN(1,I4).EQ.0) K=255-K
      IPACK=IPACK+K
      IF(I1.LT.0) IPACK=-IPACK-1
      CALL EQUAL(PACK,IPACK)
      RETURN
      END
```

```
      FUNCTION UNPACK(I,IB)
C     EXAMPLES-  IUNPAK('ABCD',1) IS 'A   '
C                IUNPAK('ABCD',2) IS 'B   '
C                IUNPAK('ABCD',3) IS 'C   '  ETC.
C     IN WORD I, IB IS BYTE 1,2,3, OR 4, OTHERWISE 1 IS ASSUMED.
C     THE RESULT IS ALWAYS RETURNED IN BYTE 1, WITH BLANKS TO THE RIGHT.
C     PROGRAMMED OCT/68 BY D. FLATHMAN
      ENTRY IUNPAK(I,IB)
      K=I
      IF(I.LT.0) K=-(I+1)
      GO TO (1,2,3,4),IB
    1 K=K/16777216
      IF(K-127)5,5,8
    2 KA=K/65536
      K=KA-KA/256*256
      IF(K-127)5,5,8
    3 KA=K/256
      K=KA-KA/256*256
      IF(K-127)5,5,8
    4 K=K-K/256*256
      IF(K-127)5,5,8
    5 IUNPAK=K*16777216+4210752
      IF(I)6,7,7
    8 IUNPAK=(255-K)*16777216+4210752
      IF(I)7,6,6
    6 IUNPAK=-IUNPAK-8355712
    7 CALL EQUAL(UNPACK,IUNPAK)
      RETURN
      END
      SUBROUTINE EQUAL(Y,X)
      Y=X
      RETURN
      END
      FUNCTION IRALST(K)
      L=MADLST(K)
      CALL SETIND(IDIN(L)-1,-1,-1,L)
      IRALST=IDIN(L)
      IF(IRALST-2)2,2,1
    2 CALL MTLIST(L)
      CALL SETIND(0,-1,-1,L)
      CALL RCELL(L)
    1 RETURN
      END
      FUNCTION IDIN(K)
      INTEGER*2 ID(16000),LNKL(16000),LNKR(16000)
      COMMON LAVS,LW(10),ICONT(16000),ID,LNKL,LNKR
      IDIN=ID(K)
      RETURN
      END
      FUNCTION LLIN(K)
      INTEGER*2 ID(16000),LNKL(16000),LNKR(16000)
```

```
      COMMON LAVS,LW(10),ICONT(16000),ID,LNKL,LNKR
      LLIN=LNKL(K)
      RETURN
      END
      FUNCTION LRIN(K)
      INTEGER*2 ID(16000),LNKL(16000),LNKR(16000)
      COMMON LAVS,LW(10),ICONT(16000),ID,LNKL,LNKR
      LRIN=LNKR(K)
      RETURN
      END
      SUBROUTINE SETIND(I,LL,LR,K)
      INTEGER*2 ID(16000),LNKL(16000),LNKR(16000)
      COMMON LAVS,LW(10),ICONT(16000),ID,LNKL,LNKR
      IF(I+1)1,2,1
1     ID(K)=I
2     IF(LL+1)3,4,3
3     LNKL(K)=LL
4     IF(LR+1)5,6,5
5     LNKR(K)=LR
6     RETURN
      END
      SUBROUTINE STRIND(M,K)
      INTEGER*2 ID(16000),LNKL(16000),LNKR(16000)
      COMMON LAVS,LW(10),ICONT(16000),ID,LNKL,LNKR
      ICONT(K)=M
      RETURN
      END
      FUNCTION INHALT(K)
      INTEGER*2 ID(16000),LNKL(16000),LNKR(16000)
      COMMON LAVS,LW(10),ICONT(16000),ID,LNKL,LNKR
      INHALT=ICONT(K)
      RETURN
      END
      FUNCTION NAME(MAD)
      NAME=MAD-101058055
C     101058055 IS '9999'
      RETURN
      END
      FUNCTION MADLST(L)
      MADLST=L
      IF(MADLST)1,1,2
1     MADLST=L+101058055
2     RETURN
      END
      SUBROUTINE INITAS(N)
      COMMON LAVS,LW(10)
      LAVS=1
      CALL SETIND(0,N,12,LAVS)
      CALL STRIND(N,LAVS)
      DO 1 I=2,11
      LW(I-1)=NAME(I)
```

```
      CALL SETIND(9999,I,I,I)
1     CALL STRIND(-1,I)
      DO 2 I=12,N
      CALL SETIND(0,0,I+1,I)
2     CALL STRIND(0,I)
      CALL SETIND(0,0,0,N)
      RETURN
      END
      SUBROUTINE RCELL(L)
      COMMON LAVS
      CALL SETIND(-1,-1,L,LLIN(LAVS))
      CALL SETIND(-1,L,-1,LAVS)
      CALL SETIND(-1,-1,0,L)
      RETURN
      END
      FUNCTION LISTMT(K)
      L=MADLST(K)
      IF(LRIN(L)-L)3,4,3
4     LISTMT=0
      RETURN
3     LISTMT=-1
      RETURN
      END
      FUNCTION MTLIST(K)
      COMMON LAVS
      MTLIST=K
      L=MADLST(K)
      IF(LISTMT(L))3,4,3
3     LR=LRIN(L)
      LL=LLIN(L)
      CALL SETIND(-1,L,L,L)
      CALL SETIND(-1,-1,LR,LLIN(LAVS))
      CALL SETIND(-1,LL,-1,LAVS)
      CALL SETIND(-1,-1,0,LL)
4     RETURN
      END
      FUNCTION NUCELL(X)
      COMMON LAVS
      L=LRIN(LAVS)
      IF(L)1,5,1
1     CALL SETIND(-1,-1,LRIN(L),LAVS)
      IF(IDIN(L)-1)4,2,4
2     CALL IRALST(INHALT(L))
4     CALL SETIND(0,0,0,L)
      CALL STRIND(0,L)
      NUCELL=L
      RETURN
5     WRITE(6,6)
6     FORMAT('0AVAILABLE SPACE EXHAUSTED')
      STOP
      END
```

```
      FUNCTION LIST(K)
      LIST=NUCELL(X)
      IF(K-2)1,2,1
    1 CALL SETIND(3,LIST,LIST,LIST)
      GO TO 3
    2 CALL SETIND(2,LIST,LIST,LIST)
    3 CALL STRIND(-1,LIST)
      LIST=LIST-101058055
      IF(K/4) 4,5,4
    4 K=LIST
    5 RETURN
      END
      FUNCTION NAMTST(M)
      COMMON LAVS
      IF(M)5,1,1
    5 L=M+101058055
      IF(L)1,1,6
    6 IF(L-INHALT(LAVS))4,4,1
    4 IF(IDIN(L)-2)1,2,2
    2 IF(LRIN(LLIN(L))-L)1,3,1
    3 NAMTST=0
      RETURN
    1 NAMTST=-1
      RETURN
      END
      FUNCTION NEWTOP(M,L)
      I=MADLST(L)
      IR=NUCELL(X)
      LR=LRIN(I)
      NEWTOP=IR
      CALL SETIND(-1,-1,IR,I)
      CALL SETIND(0,I,LR,IR)
      CALL SETIND(-1,IR,-1,LR)
      IF(NAMTST(M))2,1,2
    1 CALL SETIND(1,-1,-1,IR)
      LR=MADLST(M)
      CALL SETIND(IDIN(LR)+1,-1,-1,LR)
    2 CALL STRIND(M,IR)
      RETURN
      END
      FUNCTION NEWBOT(M,L)
      I=MADLST(L)
      IL=NUCELL(X)
      LL=LLIN(I)
      NEWBOT=IL
      CALL SETIND(-1,-1,IL,LL)
      CALL SETIND(0,LL,I,IL)
      CALL SETIND(-1,IL,-1,I)
      IF(NAMTST(M))2,1,2
    1 CALL SETIND(1,-1,-1,IL)
      LL=MADLST(M)
```

```
      CALL SETIND(IDIN(LL)+1,-1,-1,LL)
2     CALL STRIND(M,IL)
      RETURN
      END
      FUNCTION IPOPUP(M,I)
      L=MADLST(I)
      K=LRIN(L)
      IF(IDIN(K)-2)1,2,2
1     NEXT=LRIN(K)
      M=INHALT(K)
      CALL SETIND(-1,-1,NEXT,L)
      CALL SETIND(-1,L,-1,NEXT)
      CALL RCELL(K)
      IPOPUP=0
      RETURN
2     IPOPUP=-1
      M=0
      RETURN
      END
      FUNCTION IPOPBT(I,L)
      K=MADLST(L)
      M=LLIN(K)
      IF(IDIN(M)-2)1,2,2
1     NEXT=LLIN(M)
      I=INHALT(M)
      CALL SETIND(-1,NEXT,-1,K)
      CALL SETIND(-1,-1,K,NEXT)
      IPOPBT=0
      CALL RCELL(M)
      RETURN
2     I=0
      IPOPBT=-1
      RETURN
      END
      FUNCTION ITOP(M,L)
      LR=LRIN(MADLST(L))
      IF(IDIN(LR)-2)1,2,2
1     M=INHALT(LR)
      ITOP=0
      RETURN
2     M=0
      ITOP=-1
      RETURN
      END
      FUNCTION MADNTP(K,N)
      MADNTP=MADLST(K)
      DO 1 I=1,N
1     MADNTP=LRIN(MADNTP)
      RETURN
      END
```