PLATO® Author Language: Part I

A course in the CREATE curriculum

Third Edition

CONTROL DATA

Pub. No. 76360726B

Copyright © 1978 by Control Data Corporation

All rights reserved. No part of this material may be reproduced by any means without permission in writing from the publisher.

Printed in the United States of America.

2 3 4 5 6 7 8 9/84 83 82 81

Acknowledgments

The Control Data Education Company wishes to acknowledge the comments and contributions of the following consultants:

Daniel E. Bailey, University of Colorado; K. Dean Black, Brigham Young University; Karen K. Block, Learning Research and Development Center, University of Pittsburgh; C. Victor Bunderson, Brigham Young University; D. Cecil Clark, Brigham Young University; Celia R. Davis, University of Illinois; Walter Dick, Florida State University; Robert Glaser, Learning Research and Development Center, University of Pittsburgh; Keith A. Hall, Pennsylvania State University; Joseph A. Klecka, University of Illinois; Jerry W. Klein, West Virginia College of Graduate Studies; Helen Koch, Minnesota Educational Computing Consortium; M. David Merrill, Brigham Young University; Gaylene R. Rosaschi, Brigham Young University; Daniel L. Stufflebeam, Western Michigan University; Karl L. Zinn, University of Michigan; and The Center for Educational Design at Florida State University, which contributed significantly to portions of an earlier edition of this curriculum.

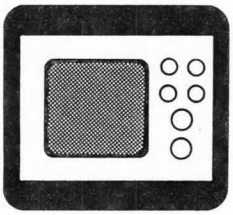
Visual Cues in *CREATE*

These visual cues at the beginning of each learning activity in the *CREATE* curriculum indicate the media used for instruction:

TEXT



VIDEOTAPE



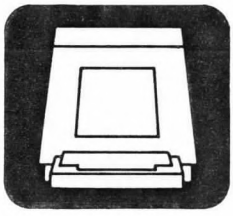
TEXT AND EXERCISE



FILMSTRIP



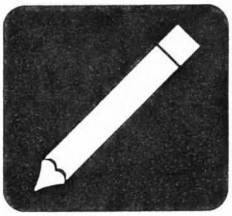
COMPUTER-ASSISTED INSTRUCTION



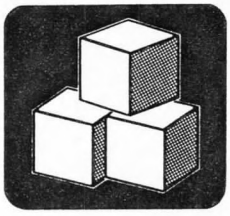
AUDIOTAPE



EXERCISE



PROJECT



Contents

PLATO AUTHOR LANGUAGE: PART I COURSE INTRODUCTION 1

Unit 1. Introduction 3
1-A. Course Introduction 5
1-B. Introduction to the Control Data PLATO System 13
1-C. Capabilities of the Control Data PLATO System 15

Unit 2. Programming Fundamentals 18
2-A. Getting Started: Development Documents 20
2-B. Screen Layout 26
2-C. Instruction Format 30
2-D. Instruction Types, Basic Instructions, and Execution States 32
2-E. Lesson Execution 45
2-F. How to Read a Flowchart Representing Student Flow 47
2-G. Interactive Program 60

Unit 3. Entering and Editing Code 65
3-A. Control Data PLATO Displays 67
3-B. Author Sign-on 68
3-C. A First Look at Code 72
3-D. Text Editing, Lesson One 74
3-E. Condense Errors 77
3-F. Text Editing, Lesson Two 83
3-G. Entering Your Own Code 89
3-H. Lesson Space Structure and Extended Core Storage 90
3-I. Text Editing, Lesson Three 93
3-J. Editing Help 95

Unit 4. Lesson Sequencing and Display Control 96
4-A. Lesson Control With Function Keys 98
4-B. Help Sequences 106
4-C. Author-Initiated Branching 116
4-D. The Helpop Family 122
4-E. Display Control Instructions 124
4-F. Comprehensive Practice 130

Unit 5. Information and Communication 151
5-A. The *Control Data PLATO Author Language Reference Manual* 153
5-B. The On-line Reference: *Aids* 155
5-C. The TERM-talk Option 159
5-D. The Users List and Talk Options 161
5-E. The TERM-consult Option 163
5-F. PLATO Notes 165

Unit 6. Variables, Calculations, and Conditional Instructions 167
6-A. Variables 169
6-B. PLATO Variables 175
6-C. Variables: Assigning Values, Assigning Names, and Displaying 178
6-D. Random Number Generation 188
6-E. Conditional Instructions: Writing, Judging, and Calculating 194
6-F. Conditional Instructions: Branching Between Units 208
6-G. Conditional Instructions: Branching Within Units 216
6-H. Pause Processing 236
6-I. Programming With Variables, Calculations, and Conditional Instructions 244

Unit 7. Graphics 256
7-A. The Fine Grid 258
7-B. Graphics Instructions 262
7-C. TERM-cursor 282
7-D. Insert and Show Display 283
7-E. Programming With Graphics 285

Unit 8. Additional Judging Instructions 292
8-A. Storing Student Responses 294
8-B. Matching Anticipated Responses 306
8-C. Judging Numerical Responses 313

PLATO Author Language: Part I
Course Introduction

Think of a PLATO lesson as a jigsaw puzzle; the CAI designer/developer determines the overall picture to be created; the photo on the cover of the puzzle box is analogous to the general design of the displays the student will see. The programmer decides which pieces to use and how to put all the pieces together to create the specified result.

The "pieces" of a computer-assisted instruction (CAI) lesson are called *instructions*. They tell the computer what to do, and where or how to do it. Although the PLATO Author Language consists of numerous instructions, each important in its own way, you don't need to know *all* of the instructions to begin to program CAI lessons. *PLATO Author Language: Part I* explains how to use those instructions the designers of this course feel are used most often by PLATO programmers. This course contains eight units:

Unit 1, "Introduction," explains how to proceed through this course, presents terminology used in discussing the PLATO system, and identifies capabilities of the PLATO system.

Unit 2, "Programming Fundamentals," presents topics fundamental to programming: terminology; CAI development documents; instruction types, purposes, and formats; execution states; and flowcharts.

Unit 3, "Entering and Editing Code," explains how to enter and edit code using the PLATO system.

Unit 4, "Lesson Sequencing and Display Control," discusses instructions used to sequence and manipulate displays.

Unit 5, "Information and Communication," presents ways to obtain information about the PLATO Author Language.

Unit 6, "Variables, Calculations, and Conditional Instructions," discusses instructions that can produce different effects, depending on a variety of factors.

Unit 7, "Graphics," explains how to create displays using different sizes and orientations of text, lines, and geometric figures.

Unit 8, "Additional Judging Instructions," discusses instructions that allow you to store unanticipated student responses, record which anticipated responses are made, and judge numerical responses.

This course is designed to train you to code PLATO lessons from programmable-ready materials. It does *not* provide instruction in computer-assisted instruction (CAI) design principles. If you must also learn to design your CAI activities, you should plan to take the *CAI Design and Development* course, which covers CAI design.

Before you begin this module, ask your learning center technician for audiotape ATA61a, "Introduction to PLATO Author Language: Part I." This audiotape introduces the *PLATO Author Language: Part I* course.

1

Introduction

This unit explains how to proceed through this course, *PLATO Author Language: Part I.* It also explains the terminology used in discussing the PLATO system, and identifies the PLATO system's capabilities.

INSTRUCTIONAL OBJECTIVES

After completing this unit, you should be able to:

- Determine how the organization of this course contributes to its effectiveness as an individualized, computer-based education curriculum
- Interact with the PLATO system to receive instruction
- Interact with the PLATO system to take tests
- Register for a course and sign on to the PLATO system
- Identify the steps to follow in completing a course
- Recognize the meanings of terms associated with the PLATO system
- Identify current capabilities of the PLATO system

LEARNING ACTIVITIES

_____ 1-A. Course Introduction. Text/CAI: If this is your first *CREATE* course, be sure to complete this activity. It explains the organization of your course. It also explains how to interact with the PLATO system to receive instruction and take tests, and how to register for your course. Finally, it lists the step-by-step process

to follow in completing your course. If you have already completed the *Introduction* module of the *CREATE Fundamentals* course, you may either bypass this activity or take it again for review.

_____ 1-B. Introduction to the Control Data PLATO System. Text: This activity defines terms associated with the PLATO system and identifies ways in which it can be used effectively.

_____ 1-C. Capabilities of the Control Data PLATO System. CAI/Exercise: This activity provides examples of how the PLATO system can provide text displays, graphics, response judging, feedback, random generation, and record keeping. If you have already completed the *Computers in Instruction* module of the *CREATE Fundamentals* course, you may either bypass this activity or take it again for review.

WHERE TO BEGIN

IF YOU HAVE COMPLETED THE *INTRODUCTION* MODULE OF THE *CREATE FUNDAMENTALS* COURSE, BYPASS ACTIVITY 1-A AND SIGN ON TO A PLATO TERMINAL. Your learning center technician will give you your sign-on name and the group name. If you are very familiar with the PLATO system, you may wish to test out of some activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.

IF YOU HAVE NOT COMPLETED THE *INTRODUCTION* MODULE OF THE *CREATE FUNDAMENTALS* COURSE, TURN DIRECTLY TO ACTIVITY 1-A OF THIS UNIT. When you complete activity 1-A, sign on to a PLATO terminal. Your learning center technician will give you your sign-on name and the group name. If you are familiar with the PLATO system, you may wish to test out of some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.



1-A. COURSE INTRODUCTION

Determine how the organization of this course contributes to its effectiveness as an individualized, computer-based education curriculum.

Interact with the PLATO system to receive instruction.

Interact with the PLATO system to take tests.

Register for a course and sign on to the PLATO system.

Identify the steps to follow in completing a course.

ORGANIZATION OF THE COURSE

PLATO Author Language: Part I is composed of units, each of which is composed of learning activities. You are reading the first learning activity in the first unit of the course.

The purpose of this organization is to facilitate the individualization of your instruction by allowing you to receive only the instruction you need. This is the way it works: For each unit, you will be tested on the material presented in the learning activities within the unit. On the basis of the pretest results, you will be given a list of those learning activities that cover areas in which you need instruction. You take only the prescribed learning activities. Upon completion of the learning activities, you will take a test. Passing the test means that you know the material in the unit well enough to go on to the next unit.

Incidentally, not all of the learning activities are printed text readings. Some combine a text reading with a computer-assisted instruction (CAI) lesson. For example, this activity is both CAI and text. Other activities involve writing and executing PLATO Author Language programs. In addition to using the PLATO system to receive instruction and practice in programming, you will use it to take tests and receive your lists of prescribed learning activities.

INTERACTING WITH THE PLATO SYSTEM

You will be using the PLATO system for instructional and testing purposes. The CAI lessons in this activity explain how to use this computer system. They also give you your first opportunity to interact with the PLATO system in this course.

This CAI activity contains three parts, or lessons:

- "Introduction to the Control Data PLATO Terminal." Be sure to take this lesson if you have had no previous experience using the terminal.
- "Sample Control Data PLATO Instruction." Select this lesson if you wish to experiment with an example of a computer-assisted instruction (CAI) activity.
- "Testing on Control Data PLATO." Be sure to take this lesson if you are unfamiliar with the computer-managed instruction (CMI) testing system. It explains how you will use the PLATO system to take tests and how you can determine which units and learning activities you need to take.

Now ask your learning center technician to help you locate a Control Data PLATO terminal and get started on this CAI activity. When you finish these lessons, return to this text activity.

SIGNING ON TO CONTROL DATA PLATO

You are a busy person whose instructional schedule should not have to depend on the schedules of others. You had to depend on the learning center technician to start you on your first CAI activity. After this activity, with the exception of course registration, you will be signing on to Control Data PLATO tests and activities on your own. To do this, you must know the proper sign-on procedures.

You will need a few key words or phrases that will allow you to use the PLATO tests and activities included in your course. Your learning center technician will help you determine some of these words and will provide you with others. These words include:

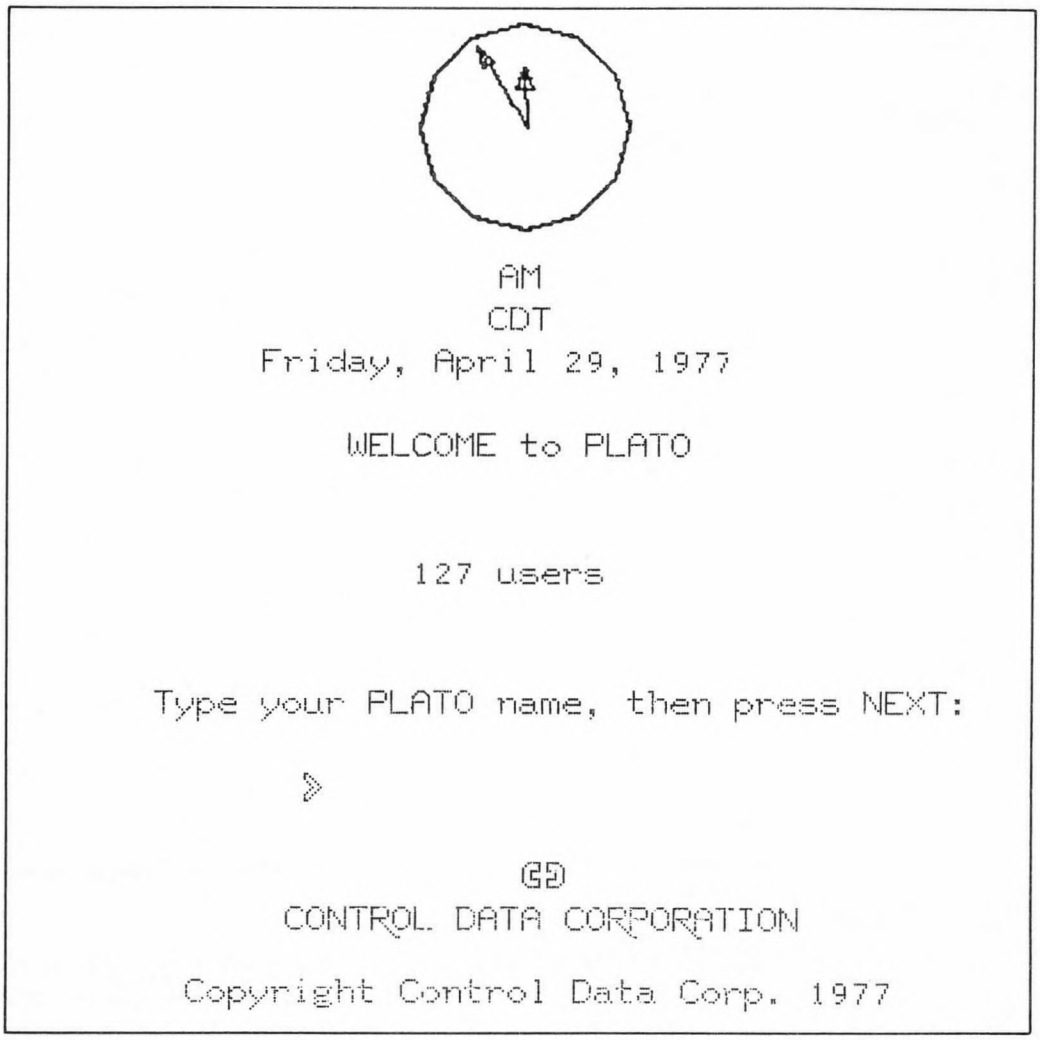
- The sign-on name you select for yourself
- The group name for your specific course
- Your password, which will allow you, and only you, to access the tests you need

The following pages represent the first five displays that will appear on the computer terminal's screen every time you sign on. Each display after the first requires you to type in a specific word or phrase. See your learning center technician now to get your sign-on information.



Step 1. If the screen displays anything but this message, press the SHIFT key and the STOP key (hold SHIFT down and press STOP) several times until the message appears. If the message still does not appear, press BACK several times.

Step 2. When this message is displayed, press the NEXT key.



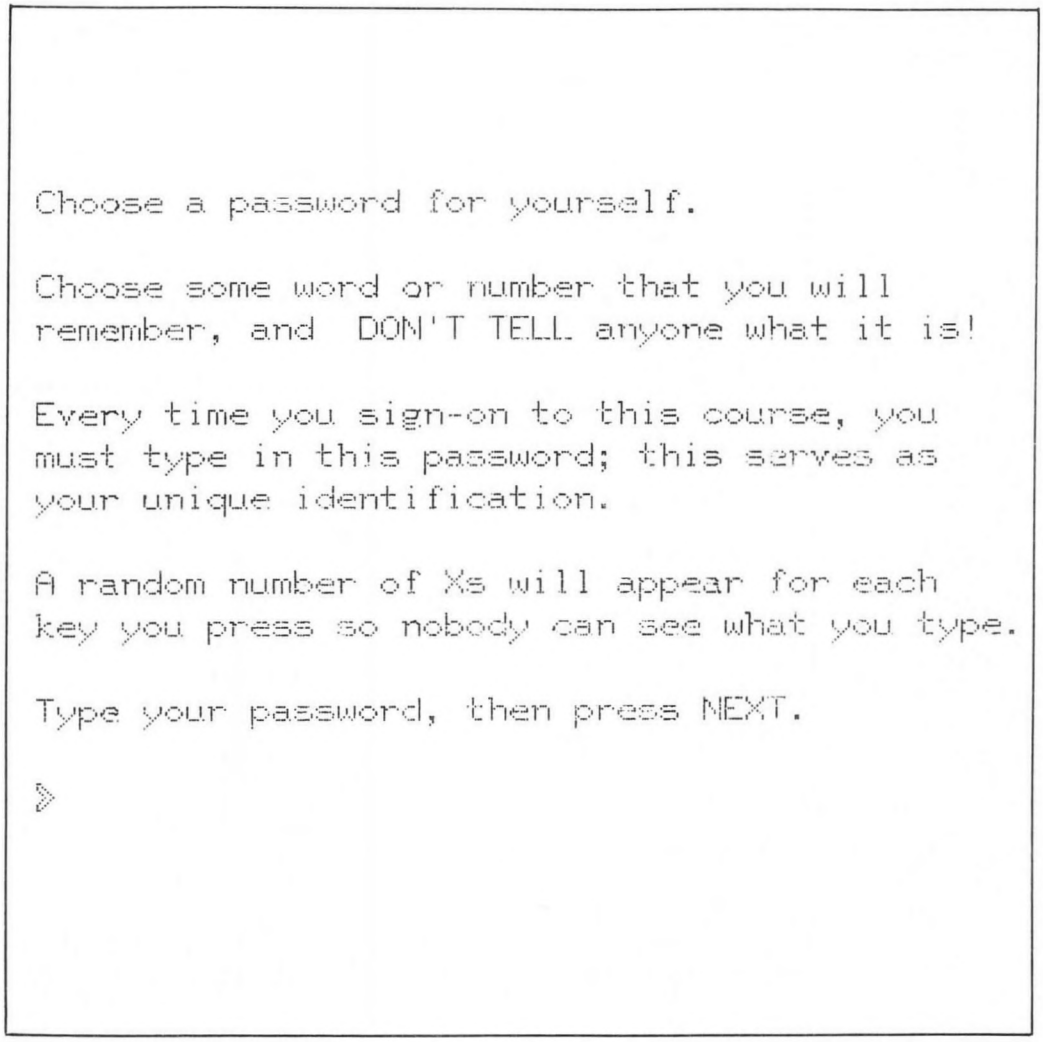
Step 3. When you see this display, type your sign-on name, then press the NEXT key.

Record your sign-on name in the appropriate space in the diagram.



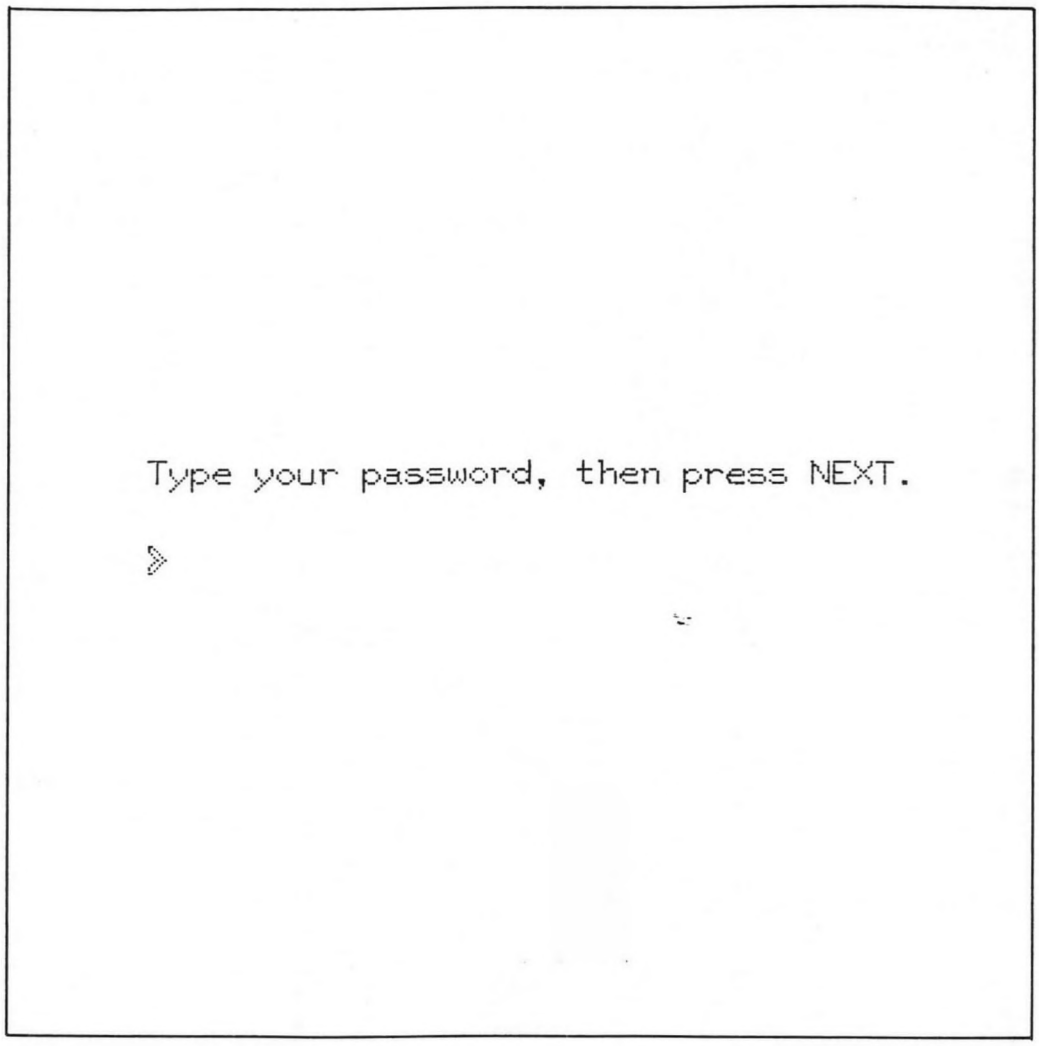
Step 4. When you see this display, type the name of your group, then hold down the SHIFT key and press the STOP key *simultaneously.*

Record your group name in the appropriate space in the diagram.



Step 5. You will see this display only when you register for a course. It asks you to choose your password for the course and to type it twice. This completes the registration procedure.

Record your password for the PLATO Author Language course in the appropriate space in the diagram.



Step 6. After you have registered for a course, you will sign on to the PLATO system in the same way, with the exception of the password display. When signing on, type your password only once when you see this display.

Record your password in the appropriate space in the diagram.

Step 7. You should now see the first CMI display. Follow the directions given on the display.

STEPS IN COMPLETING THE COURSE

The following list summarizes the steps you take in completing the course.

1. Sign on to the terminal to take the test and receive the assignments (prescribed activities) for the first unit of this course (unit *a*) *or* skip the test and go directly to the assignments (prescribed activities).
2. In the blanks beside the activity descriptions in the text introduction to this unit, check off the activities prescribed to you.
3. Complete the prescribed activities. Before beginning a prescribed CAI activity, read the activity introduction in your text.
4. When you have completed all assigned activities, return to the terminal to take the test. If you do not pass all sections of the test, complete the assigned activities. Repeat this process until you have passed all sections of the test.
5. When you have passed all sections of the test for the unit, read the text introduction to the next unit.
6. Repeat steps 1 through 5 for each unit of the course. Sign on to the terminal now and take the test or go directly to the assignments for unit *a*.

TESTING DIRECTIONS

Now that you know how to use the PLATO system and complete a course, sign on to a PLATO terminal. If you are very familiar with the PLATO system, you may wish to test out of some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.



1-B. INTRODUCTION TO THE CONTROL DATA PLATO SYSTEM

Recognize the meanings of terms associated with the PLATO system.

What is the Control Data PLATO system? How is the PLATO terminal related to the computer? What makes information appear on the screen of the PLATO terminal? Although it is not absolutely necessary, it will be helpful if you know the answers to these questions, at least on a general level. The situation is similar to that faced by the beginning driver; a person does not need to know the details of the operation of an automobile in order to drive, but it is nice to have a general idea of how it works. This activity provides a general idea of how the Control Data PLATO system works.

The Control Data PLATO system consists of the PLATO terminal, a communication link, a computer, and programs written in PLATO Author Language.

PLATO TERMINAL

The PLATO terminal is a key component of the PLATO computer-based education system. Information is communicated to the student through use of a display screen. The screen may display words, graphic diagrams, or combinations of words and graphics. In addition to the displays, information may also be provided through an optional audio device. Student replies or data input are entered on a typewriter style keyboard or by an optional touch panel on the screen.

COMMUNICATION LINK

PLATO terminals must communicate with a computer for lesson information. This communication may be by a telephone line or it may be by a permanent connection between the terminal and the computer.

COMPUTER

The computer receives information from the PLATO terminal, stores information and performs calculations and comparisons, and causes the display of words and graphics on the PLATO terminal display screen. The Control Data PLATO system requires a large computer system such as the CYBER 170.

PROGRAM AND PROGRAMMING LANGUAGE

In order to receive information from the terminal, store information and perform calculations and comparisons, and cause the display of words and graphics on the terminal's display screen, the computer must be given a set of *instructions* called a *program*. The computer can do only what it is told to do. The instructions given to the computer must, however, be written in a language it can interpret. This language is called a *programming language*. For most purposes, the PLATO computer system interprets only one programming language: the PLATO Author Language.

Before it can actually follow the instructions in a PLATO Author Language program, however, the computer must transform the instructions into a language that can be executed more easily by the computer. The transformation is called *condensing*. The reason two languages are necessary is that although the computer actually uses instructions in condensed form, it is much easier to write programs in PLATO Author Language.



1-C. CAPABILITIES OF THE CONTROL DATA PLATO SYSTEM

Identify current capabilities of the PLATO system.

Computer systems can offer a great deal to the education process. But not all systems are the same; each has its own strengths and limitations. Because you are working with Control Data's PLATO system, it will be useful for you to know more about the system's capabilities.

CONTROL DATA PLATO FEATURES

Some features of the PLATO system are easily demonstrated; others are not. The system is capable of drawing pictures and scenes, and it can also animate characters. It can write in script and foreign alphabets. With the appropriate attachment, it allows students to indicate their answers by touching the computer terminal screen.

There are several other features that aid instruction. CAI displays can be monitored to allow the instructor to determine whether students are having trouble, or to check their progress in the lesson. When it is desired, students are able to communicate with each other via the terminal. All of these features contribute to the delivery of effective instruction. If you have already completed the *Computers In Instruction* module of the *CREATE Fundamentals* course, you may bypass the following CAI activity or take it again for review. If you have *not* completed *Computers In Instruction*, complete this activity now.

CAI INTRODUCTION

This CAI activity will take twenty to thirty minutes to complete. In it, you will see examples of how Control Data's PLATO system can provide the following:

- Text presentation
- Graphics
- Response judging
- Feedback
- Random generation
- Record keeping

This activity will suggest a number of educational uses for these various capabilities. If you want to write down a few of these application ideas, bring paper and pencil to the terminal with you.

Now sign on to a Control Data PLATO terminal. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "CAI Capabilities: Control Data PLATO," press LAB to take the lesson. Complete the following exercise while you do the activity.

CHECK YOUR UNDERSTANDING

Directions: Supply the information requested in each of the following sections.

List at least four specific text capabilities that Control Data's PLATO system can perform.

1. _____

2. _____

3. _____

4. _____

List at least three types of graphics that Control Data's PLATO system is capable of presenting.

1. _____

2. _____

3. _____

List at least four other capabilities of Control Data's PLATO system that are not regarded as graphics or text.

1. _____

2. _____

3. _____

4. _____

Answers

Capabilities of the PLATO system that concern text:

1. Speed of writing text
2. Sized writing–large or small
3. Related writing
4. Underlining, boxing in of important points
5. Writing of special characters and many languages
6. Writing in script
7. Rewrites over previous writing
8. Partial erase of screen
9. Full erase of screen

Capabilities of the PLATO system that concern graphics:

1. Illustrative pictures
2. Special characters
3. Animation of special characters
4. Static graphs
5. Dynamic graphs
6. Artwork

Other PLATO capabilities:

1. Sophisticated response judging (words, phrases)
2. Sophisticated feedback
3. Calculating abilities
4. Random generation
5. Student records (name, course, lesson, unit, time, date)
6. Audio
7. Slides
8. Monitoring of students' displays
9. Interterminal communication capabilities
10. Easy authoring language

2

Programming Fundamentals

This unit discusses topics fundamental to programming: CAI development documents; programming duties; instruction types, purposes, formats; execution states; and flowcharts.

INSTRUCTIONAL OBJECTIVES

When you have completed this unit, you should be able to:

- Identify documents prepared in the CAI development phase
- Identify duties of the programmer
- Identify coarse-grid positions
- Recognize the basic instruction format
- Classify instructions according to type of instruction
- Identify effects of nine basic instructions
- Classify instructions according to execution state
- Identify the effects of execution states on program execution
- Identify the flowcharting conventions and symbols used in *CREATE*
- Identify three general types of flowchart structures: linear, branching, and looping
- Identify correct and incorrect instruction sequence

LEARNING ACTIVITIES

_____ 2-A. Getting Started: Development Documents. Text: This activity identifies documents prepared in the development phase and lists the duties of the programmer.

_____ 2-B. Screen Layout. Text/Exercise: This activity explains how to identify and specify coarse-grid positions.

_____ 2-C. Instruction Format. Text: This activity identifies the basic instruction format.

_____ 2-D. Instruction Types, Basic Instructions, and Execution States. Text: This activity classifies instructions into three types, discusses the purposes and formats of nine basic instructions, and classifies instructions according to the state in which they are executed.

_____ 2-E. Lesson Execution. CAI: This activity discusses how execution states affect program execution.

_____ 2-F. How to Read a Flowchart Representing Student Flow. Text/Exercise: This activity discusses flowcharting conventions and symbols used in *CREATE*. It also identifies three types of flowchart structures: linear, branching, and looping.

_____ 2-G. Interactive Program. Text/CAI: In this activity, you will write a program and experiment with instruction sequences based on directions indicated in design documents.

TESTING DIRECTIONS

Now sign on to a Control Data PLATO terminal. If you are very familiar with the subject of programming fundamentals, you may wish to test out of some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.

2-A. GETTING STARTED: DEVELOPMENT DOCUMENTS



Identify documents prepared in the CAI development phase.

Identify duties of the programmer.

In this course, the term *programmer* refers to a person who codes lessons in a format appropriate for execution by the PLATO system. Before a programmer can write code, however, documents from the development phase must be provided; the programmer must know the structure and content of the lesson to be programmed. This activity discusses the documents produced in the development phase. It also discusses the duties of the programmer in terms of the steps that must be followed in programming a lesson.

DEVELOPMENT DOCUMENTS

The CAI development phase should provide the programmer with the following documents:

- A goal statement, which explains the general intent of the lesson
- A storybook, which tells how the student is to progress through the lesson, and generally what the student will see, including all text, student feedback, directions, record-keeping strategies, and rough charting of major displays
- A flowchart of the student paths through the lesson, including the HELP sequences and other alternatives

Figures 1 through 3 contain examples of these documents for a lesson in a course on collecting overdue accounts. The storybook is not complete; it is meant only to give you an idea of what a storybook contains. Don't worry if you are unfamiliar with some of the terms in the flowchart; these terms are specific to this lesson. You will learn more about reading flowcharts in activity 2-F.

PROGRAMMING DUTIES

The steps a programmer follows in writing a program based on the CAI development documents are:

1. Prepare screen layouts in final detail.
2. Write necessary PLATO Author Language instruction code.
3. Enter the code into the system.

GOAL STATEMENT

Accounts Receivable Lesson on Tones

This lesson will give the student practice in identifying the appropriate tone to be used in specific dunning communication situations. Tones can be easy and polite, firm and aggressive, or somewhere in between. The tone taken depends on a variety of factors, which will be explored in this lesson.

Figure 1. Development documents: the goal statement

STORYBOOK (NO RECORD KEEPING REQUIRED FOR THIS PORTION)

WHAT TONE SHOULD THE COLLECTOR TAKE IN THE DUNNING COMMUNICATION?

E-A-S-Y and P-O-L-I-T-E

or

F I R M and A G G R E S S I V E

or

SOMEWHERE IN BETWEEN

NEXT to continue

BACK for task index

Instructions:

write "EASY & POLITE" slowly

write "FIRM & AGGRESSIVE" boldly (double maybe??)

write "SOMEWHERE IN BETWEEN" normally

Figure 2. Development documents: the storybook

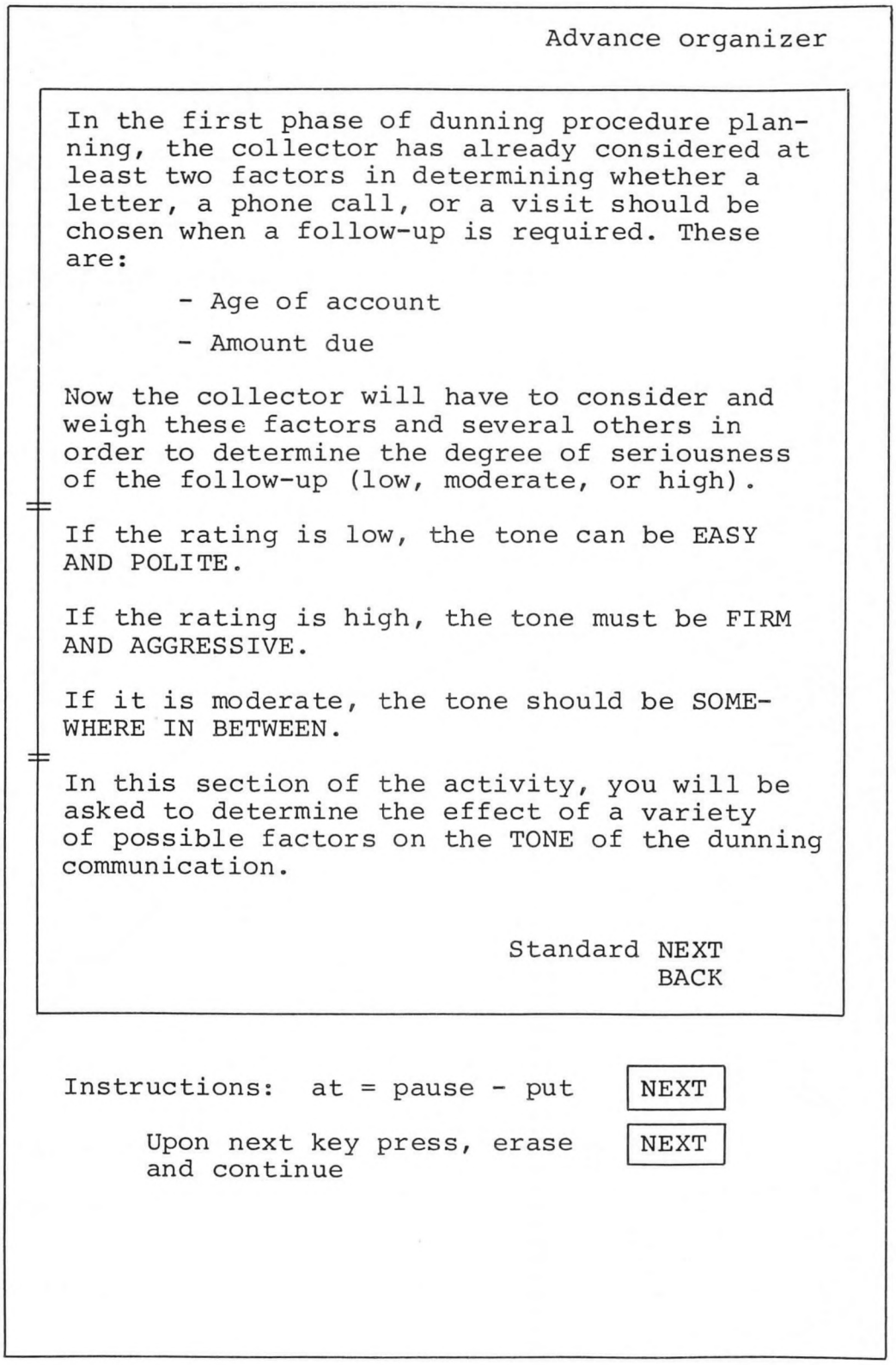


Figure 2—CONTINUED

Mini-summary

Unfortunately, real situations are rarely as simple as those you've just seen. More frequently, there are several factors to consider in any one situation. Some factors may indicate that the tone should be easy, while others indicate that it should be firm. When this happens, you must weigh the importance of each factor in the situation to arrive at the appropriate tone.

Let's look at three actual case studies to see how you might consider each factor of a case study and determine the tone of the communication.

NEXT to continue
BACK to review previous display

Figure 2—CONTINUED

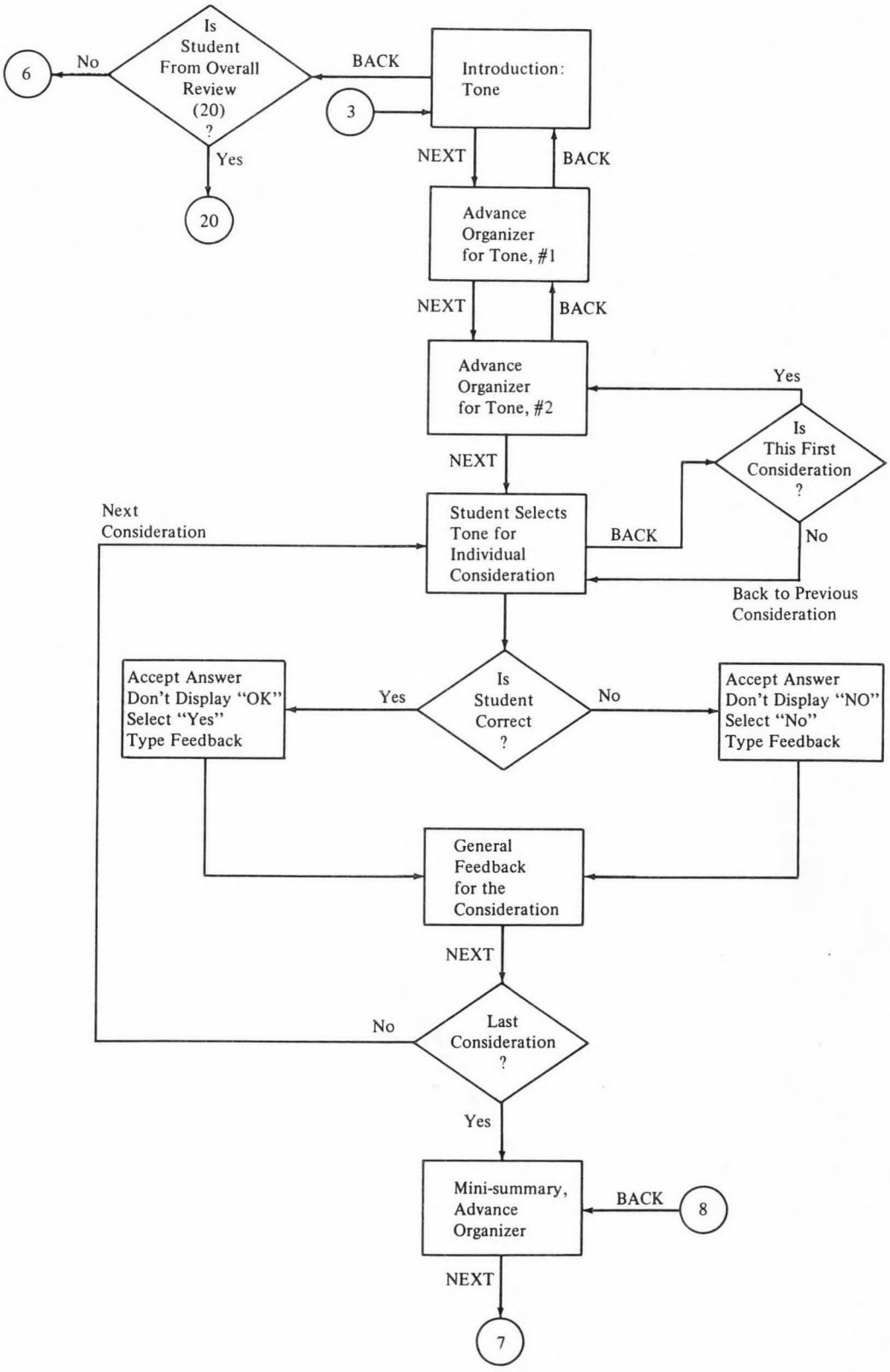


Figure 3. Development documents: flowchart of student paths

4. Have the system condense the code to identify errors.
5. Correct the errors.
6. Execute the program as a student to verify its operation.
7. Correct any errors detected in student mode.

PLATO Author Language: Part I explains the basic concepts involved in each of these steps. This listing is meant to give you an overview of the process.

SUMMARY

The programmer codes lessons according to documents supplied by the CAI developer. These documents include:

- A goal statement
- A storybook
- A flowchart of student paths through the lesson

To code a lesson, the programmer must follow the steps presented in this activity.

2-B. SCREEN LAYOUT



Identify coarse-grid positions.

The PLATO Author Language includes a large number of instructions for creating displays such as pictures, animated sequences, and simple text. Before material in any form can be displayed on the screen, however, it is necessary to specify just where on the screen it should appear. Thus, preparing the screen layout is one of the first programming steps.

There are two ways to specify screen position: coarse grid and fine grid. Coarse grid is frequently used for text display, and fine grid is used when more precise positioning is required in the display.

COARSE GRID

Coarse grid divides the screen into 32 lines with 64 positions for characters in each line. The lines are numbered from top to bottom beginning with line 1, as illustrated on the left side of figure 4.

The character positions on each line are numbered from 1 to 64, running from left to right, as indicated on the top of figure 4.

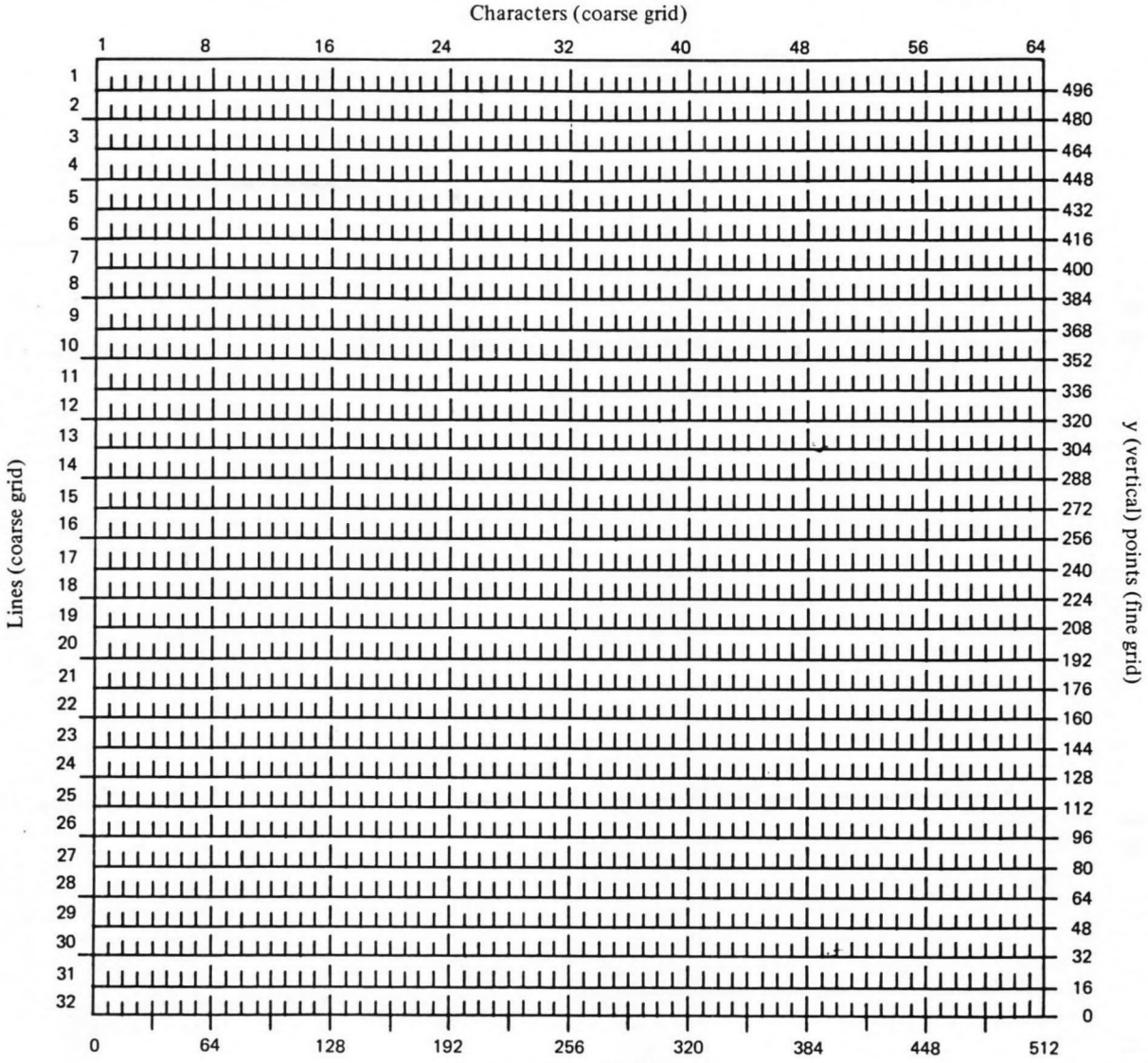
A position on the coarse grid is specified by a three- or four-digit number, with the first one or two digits specifying the line number, and the last two digits specifying the character position or column. For example, position 214 indicates line 2 and character position 14 on the line. Position 321Ø specifies line 32 and character position 1Ø on the line. Position 16Ø3 indicates line 16 and character 3 on the line. Notice the zero is required as a place holder when the specified *character* is less than 1Ø. This is not the case when the specified *line* is less than 1Ø. For example, Ø4Ø7 and 4Ø7 both indicate line 4, character 7.

FINE GRID

Positions on the fine grid are somewhat different. The screen consists of 512 rows with 512 points in each row. You will learn more about the fine grid when you learn graphics programming in a later unit.

Control Data PLATO Display Planning Sheet

Coarse grid 32 lines of 64 characters each (use numbering at top and left sides of form).
Fine grid A 512 x 512 square of points (use numbering at bottom and right sides of form).
All references are to the lower left corner of areas.



Conversions:

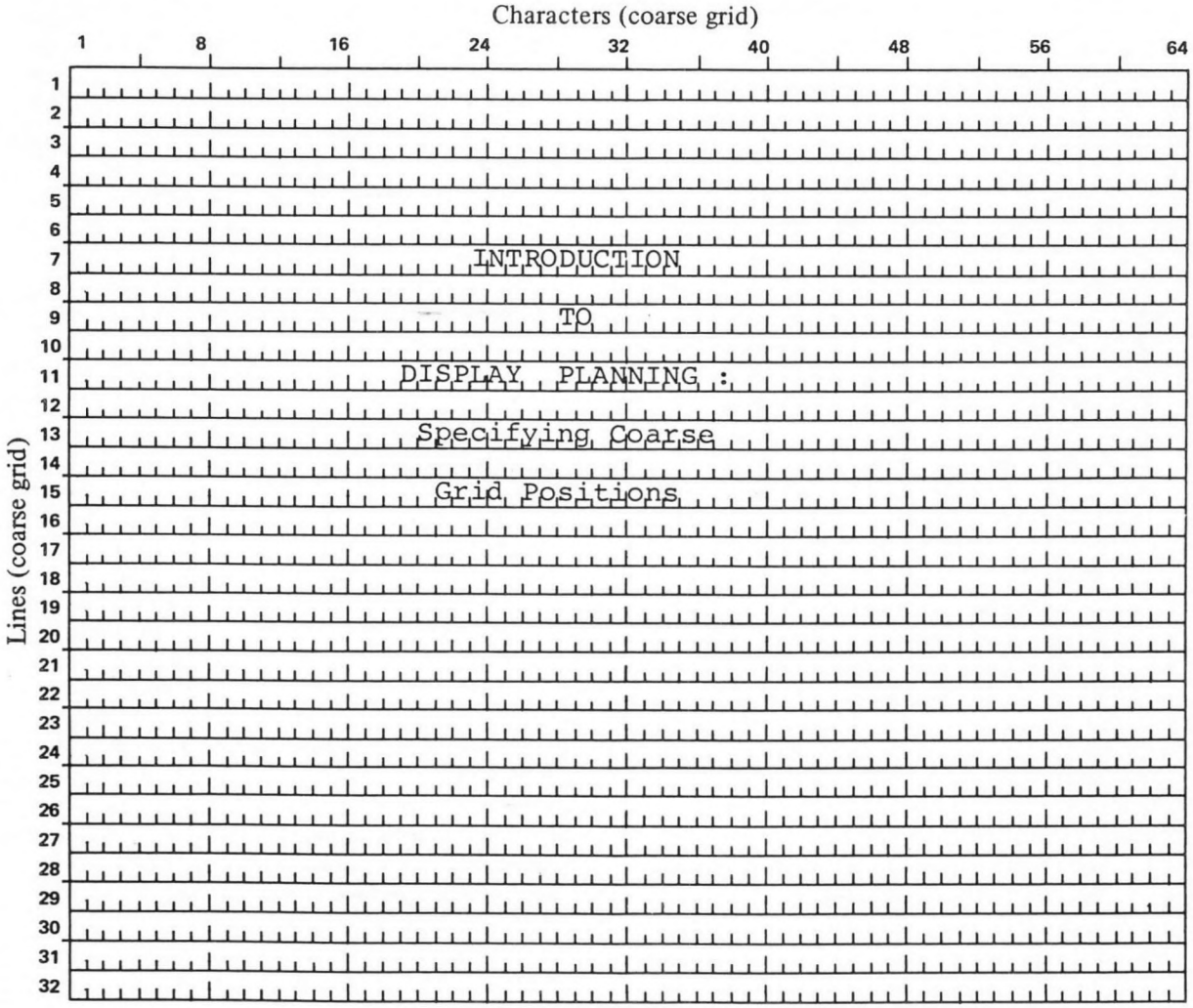
Coarse to Fine	$x = 8\ (\text{Character} - 1)$	$y = 16\ (32 - \text{Line})$
Fine to Coarse	$\text{Line} \approx 32 - \frac{y}{16}$	$\text{Character} \approx 1 + \frac{x}{8}$

Figure 4. Display planning sheet

You may already have a packet of display planning sheet forms. If you do not and you would like some, ask your learning center administration for a packet, publication number 76361198.

CHECK YOUR UNDERSTANDING

Directions: Place the correct position numbers in the spaces provided.



1. At what coarse-grid position does the word *INTRODUCTION* begin?

   _____

2. At what coarse-grid position does the word *INTRODUCTION* end?

   _____

3. At what coarse-grid position does the word *TO* begin?

   _____

4. At what coarse-grid position does the word *Coarse* begin?

   _____

5. At what coarse-grid position does the word *Coarse* end?

   _____

Answers

1. 724 or Ø724 2. 735 or Ø735 3. 929 or Ø929 4. 1332 5. 1337

SUMMARY

Screen positions can be specified using either the coarse or fine grid. The coarse grid divides the screen into 32 lines with 64 character positions in each line. Coarse-grid positions are specified with a three- or four-digit number; the first one or two digits specify the line number, and the last two digits specify the character position or column. If the character position is less than 1Ø, a zero must be included in the specification number as a placeholder. Fine grid is used for more precise positioning, such as in graphics programming, which is explained in a later unit.

2-C. INSTRUCTION FORMAT



Recognize the basic instruction format.

Most of the instructions in the PLATO Author Language use the same basic format. This activity discusses this format using two common instructions as examples.

INSTRUCTION FORMAT

Instructions consist of two parts: a command and a tag. The command portion specifies what type of instruction is to be performed and the tag specifies how or where this is to be accomplished. For example, the instruction

```
at        1223
```

has the command *at* with the tag *1223*. The instruction specifies a coarse-grid screen position for a display that will follow.

For the instruction

```
write   This is the beginning
```

write is the command, and the tag specifies what is to appear on the screen.

If a tag includes consecutive dollar signs ($$), the dollar signs and anything appearing to the right of the dollar signs are called *comments*, and are ignored when the lesson is executed. Comments can be used to make the programming for the lesson more clear to a programmer who inspects the lesson at a later date. For example:

```
at        1223  $$ position for 1st display
```

The effect of this instruction is the same as that of the previous -at- instruction. However, the addition of the comment makes it clearer what that position represents.

A comment can also be made using asterisks at the beginning of a line in which no instructions are included. For example:

```
at       1223
*** position for 1st display
```

In this case, the first asterisk must occupy the first character space, at the left-hand margin.

A third way to make a comment is to place a small letter c, followed by a space, in the left-hand margin. For example:

```
at       1223
c  position for 1st display
```

Each instruction has its own specific purpose and its own specific kind of tag. To learn the PLATO Author Language, you must learn these instructions and the way the instructions are combined into lessons. The instructions a lesson contains are referred to collectively as *source code*, *lesson code*, or simply *code*. Thus, reference is often made to the code of a lesson.

SUMMARY

PLATO Author Language instructions consist of a command and a tag. The command specifies the type of instruction to be performed and the tag specifies how or where this is to be accomplished. All instructions contained in a lesson are called *source code*, *lesson code*, or simply *code*. Comments can be included to make the programming easier to understand. Dollar signs, asterisks, or a small letter c can be used for comments.



2-D. INSTRUCTION TYPES, BASIC INSTRUCTIONS, AND EXECUTION STATES

Classify instructions according to type of instruction.

Identify effects of nine basic instructions.

Classify instructions according to execution state.

To begin programming, you should know the types of instructions that can be used, the instructions themselves, and two execution states in which the PLATO system processes instructions. This activity presents an overview of these topics.

INSTRUCTION TYPES

PLATO Author Language instructions are used to perform many types of processing. Instructions are categorized according to their purposes. In this activity, you will learn about instructions in three of the categories:

- Display instructions. These instructions allow the programmer to specify positions on the screen and to display text or graphic information.
- Lesson control instructions. These instructions are used to create the lesson structure and to control the sequence in which the users can execute the lesson.
- Response handling instructions. These instructions allow the programmer to provide for student interaction with the PLATO system. They allow the system to receive and judge the accuracy of student responses.

When you refer to the *Control Data PLATO Author Language Reference Manual* for instruction information in a later activity, you will find it organized according to these categories. Each of the instructions presented in this activity falls into one of these three categories.

BASIC INSTRUCTIONS

Obviously, you can't program a lesson without instructions. However, you only have to know a few instructions to get started. The nine instructions presented in this activity are enough to allow you to program a simple lesson. The instructions are:

-unit-
-at-
-write-
-arrow-
-answer-
-specs-
-wrong-
-no-
-endarrow-

Notice that each instruction is preceded and succeeded by a dash. The dashes are not part of the instruction; they merely set the instruction apart from other words in the text.

The -unit- Instruction

Lesson code is grouped into sections called *units*. Units can be thought of as building blocks of code that perform specified actions or displays. A PLATO lesson comprises many units, which are manipulated by the programmer to perform as specified by a designer. Nearly all lesson sequencing is accomplished using units.

The beginning of each unit is denoted by the -unit- instruction:

```
unit    NAME OF UNIT
```

The instruction tag specifies the unit name. This name may be up to eight characters in length (a capital letter counts as two characters). The characters *x* and *q* cannot be used as unit names by themselves, and no two units in a lesson may have the same name. The end of a unit is recognized when the next -unit- instruction is encountered. With the instructions you will learn in this activity, student entry into a new unit causes the previous unit display to erase. Because the -unit- instruction controls the sequence of the lesson, it is classified as a *lesson control* instruction.

In this activity, you will see how instructions combine to build a unit. This unit is named *birch*. Thus, the instruction is written:

```
unit    birch
```

The -at- Instruction

The -at- instruction is classified as a *display* instruction because it specifies the screen location at which the display will start. It uses this format when used to specify a coarse-grid location:

```
at      COARSE GRID LOCATION
```

For example,

```
at      Ø515
```

specifies line 5 (from the top), starting with character position 15.

The -write- Instruction

The -write- instruction is a *display* instruction; it displays text material on the PLATO screen. The format for this instruction is:

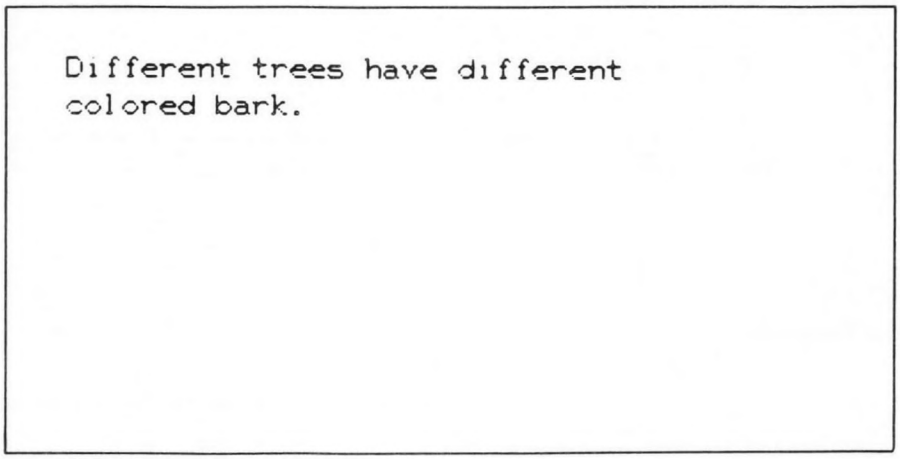
```
write   MESSAGE TO BE DISPLAYED
```

The display starts at the screen position specified by the previous -at- instruction. The message tag can be more than one line in length. For example:

Line Number

```
1       unit    birch
2       at      Ø515
3       write   Different trees have different
4               colored bark.
```

This source code results in this display because the second line of the -write- tag begins on a new line:



If you reach the end of a line when typing your -write- message, you can continue typing. The line will "wrap around" and continue on the line below in the command field, but won't be counted as a line. For example, these instructions

```
1   unit    wrap

2   at      5Ø5

3   write   Let's see how far we can go before this line wraps a
    round, and let's see what it will do to the display.
```

will look like this on the student's display:

```
Let's see how far we can go before this line wraps around, a.
nd let's see what it will do to the display.
```

Thus, you will have to estimate the appropriate number of characters you can write on a line to create an acceptable display. Your display planning sheet can help at this point; as you become more experienced in programming, you will develop a "feel" for line length.

If no -at- instruction precedes a subsequent -write- instruction, the display will start following the last character of the previous -write- instruction. Thus, the code

```
unit    birch
at      Ø515
write   Different trees have different
        colored bark.
write   Which tree has white bark?
```

appears on the screen as:

```
Different trees have different
colored bark.Which tree has white bark?
```

If no -at- instruction is ever specified, the first -write- display of each unit will begin at screen location 1Ø1.

The -arrow- Instruction

The request for a student response is initiated by the use of the -arrow- instruction. It places an arrow at the point on the screen where what is called the *arrow*

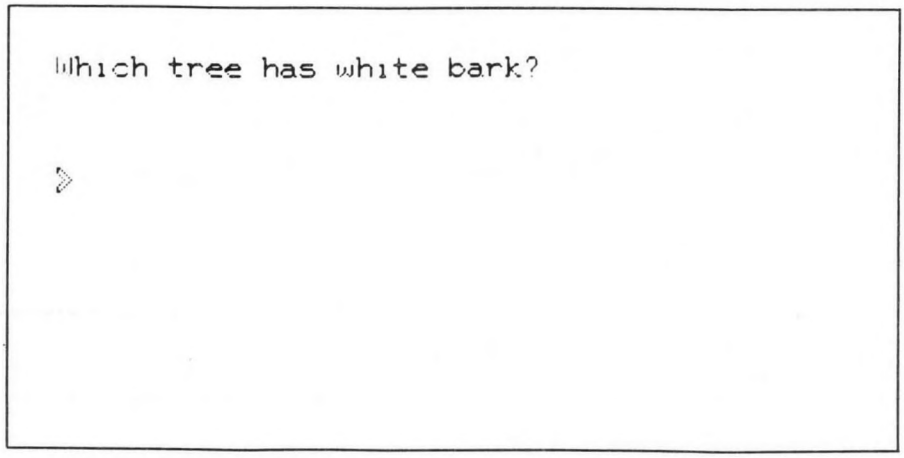
processing begins. Arrow processing includes getting and judging the student response. Thus, it is classified as a *response handling* instruction. This -arrow- instruction specifies a coarse-grid location:

```
arrow   1515
```

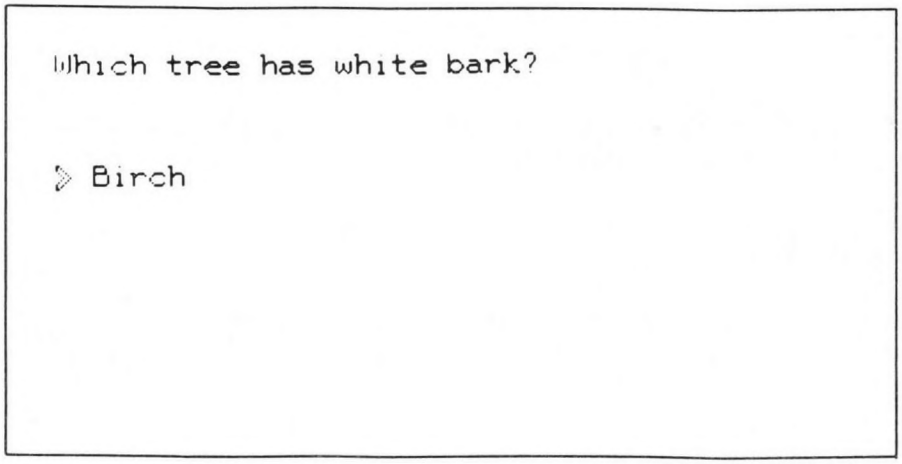
The source code

```
unit    birch
at      Ø515
write   Which tree has white bark?
arrow   Ø815
```

will appear on the screen as:



If the student types in *Birch*, this word will appear to the right of the arrow:



The -answer- Instruction

One way to judge a student response as *correct* is by using the -answer- instruction:

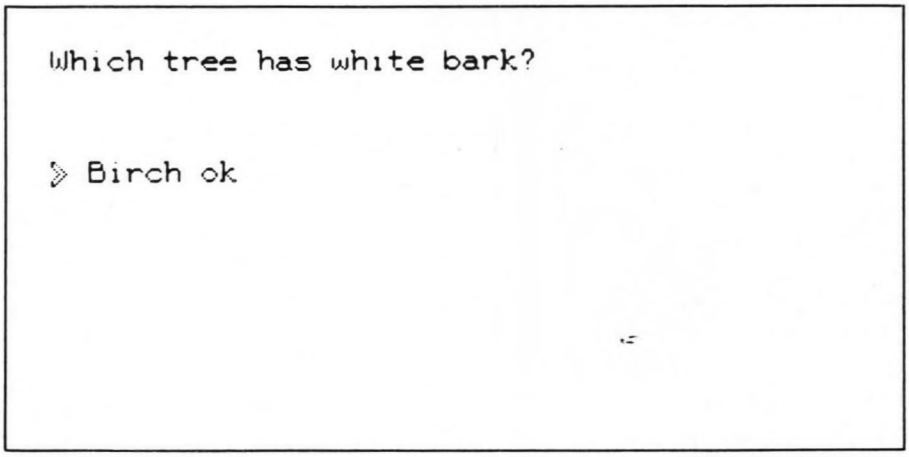


The -answer- instruction is classified as a *response handling* instruction.

When the student enters a response on the keyboard and presses the NEXT key, the response is compared to the -answer- instruction tag. For example, if the code is

```
unit     birch
at       Ø515
write    Which tree has white bark?
arrow    Ø815
answer   Birch
```

the student response of *Birch* will be judged correct and the letters *ok* will appear beside the response:



If extra words are not specified in the brackets of the instruction, the student response must match the answer instruction tag exactly. If the response does not match, the PLATO system underlines extra words with xxx s, misspelled words with === signs, words out of order with ⇔ signs, missing capital letters with ↑ signs, and missing words with Δ signs. For example, if the student entered the response, *It is a birch*, that answer would be judged incorrect:



If another word with the same or similar meaning can be entered as a response, synonomous words to be accepted can be enclosed in parentheses. For example,

```
answer   <its,it's,is,a,the>(Birch,Poplar)
```

will also accept *Poplar* as a substitute for *Birch*. For the answer to be judged correct, one of these two answers *must* be entered by the student.

When the synonyms include phrases, the words in the phrase must be separated by asterisks. In the example (Birch, Poplar, fraxinus*ash), the words in the phrase are separated by an asterisk.

The -specs- Instruction

It is important to note that if the answer specified is capitalized, the response must also be capitalized to be judged correct. This may not always suit your purposes. The -specs- instruction allows you to modify the student response before it is judged. The format for the instruction is:

```
specs   LIST OF OPTIONS DESIRED
```

The instruction is placed after the arrow and before the response handling instructions that require the modification. The modification of the responses in the arrow remains in effect until that arrow is satisfied, or until a new -specs- instruction is executed.

Some commonly used -specs- options are:

- *bumpshift*: Shift codes (capitals) in the response are removed.
- *okextra*: Extra words are permitted in the response.
- *okspell*: Recognizable misspellings are permitted.

For example,

```
specs    bumpshift,okextra,okspell
answer   <it7s>(birch,poplar)
```

will accept any of these responses, and others, as correct:

```
» birch
» it is a birch tree
» birch tree
» Birch tree
» Poplar tree
» It's the birch tree
```

Note that the acceptable answers specified in the answer instruction (birch,poplar) are *no longer capitalized*. If bumpshift is specified and your answer tag contains capital letters, the student response will never match, even if the student uses capital letters; bumpshift will cause capital letters in the student response to be bumped before the response is judged.

In this section of code

```
arrow    1010
specs    bumpshift,okextra,okspell
answer   (Birch,Poplar)
```

the student would enter a response at the arrow provided on the display. Assume the response is capitalized. The PLATO system then executes the -specs- instruction and "realizes" that the student's capital should be "bumped." The PLATO system then tries to find a match for the *un*capitalized response. Because the tag in the -answer- instruction is capitalized, no match is ever found.

Note the *7* in the optional word *it7s* in this section of code:

```
specs    bumpshift
answer   <it7s>(birch,poplar)
```

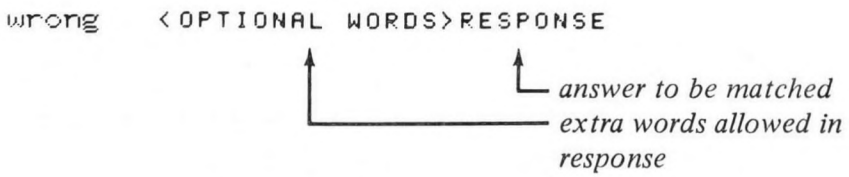
The *7* is the lower case form of the apostrophe (the key on the terminal looks like this:

). The PLATO system would consider the apostrophe a capital in this case, and would bump it from the student response. The -specs- instructions are classified as *response handling* instructions because they are used to process the student's response.

The -wrong- Instruction

The -wrong- instruction allows the author to give the students a message (feedback) for *specific* wrong answers. It is classified as a *response handling* instruction because of its connection with student responses.

The format for the -wrong- instruction is:



This instruction operates in the same way as the -answer- instruction except that the word *no* is written to the right of the student response, and the student is required to try again. After the wrong answer is recognized, any -write- instruction following this -wrong- instruction will be displayed. When a correct response is received, the -wrong- instruction is ignored. An example of the use of the -wrong- instruction is:

```
unit     birch
at       Ø515
write    Which tree has white bark?
arrow    Ø815
specs    bumpshift,okextra,okspell
answer   (birch,poplar)
wrong    pine
at       1Ø15
write    No, a pine tree has brown bark.
```

If the student answers *the pine tree*, this code will appear on the screen as:

```
Which tree has white bark?

> the pine tree no

No, a pine tree has brown bark.
```

In this case, the feedback for the wrong answer appears two lines below the arrow, as specified by the -at- instruction tag (1Ø15). If the position is not specified, the PLATO system automatically places the feedback *three* lines below the student response.

When the student presses the NEXT key, the response, the *no*, and the feedback are erased. The student is required to answer correctly before proceeding.

More than one wrong answer can be specified in a -wrong- instruction. For example:

```
unit     birch
at       Ø515
write    Which tree has white bark?
arrow    Ø815
specs    bumpshift,okextra,okspell
answer   (birch,poplar)
wrong    (pine,maple,oak)
at       1Ø15
write    The tree you've named has brown bark.
```

If the student responded with *maple tree*, the display would appear as:

```
Which tree has white bark?


» maple tree no

The tree you've named has brown bark.
```

Notice that the feedback specified in the -write- instruction is appropriate for all anticipated incorrect answers.

The -no- Instruction

In the previous examples, the student replies must match the -answer- or -wrong- commands exactly if the student is to receive feedback. The -no- instruction allows execution of regular commands when the student enters an incorrect

response *other than the anticipated incorrect responses*. Thus, the -no- instruction can be used to present feedback to enable the student to respond with the correct answer rather than just to judge the response as incorrect with no feedback. For example, the code

```
unit     birch
at       Ø515
write    Which tree has white bark?
arrow    Ø815
specs    bumpshift
answer   (birch,poplar)
wrong    pine
at       1Ø15
write    No, a pine tree has brown bark.
no
at       1Ø15
write    It is a birch tree.
```

provides the student with the correct answer in the event that no correct entry is made and matched. The student must now enter the correct answer and press NEXT for the lesson to continue. The -no- instruction does not use a tag. Its format is:

```
no        BLANK TAG
```

The -no- instruction is classified as a *response handling* instruction.

The -endarrow- Instruction

If you wish to ask more than one question in a unit, you should use the -endarrow- to mark the end of the processing of the first arrow. This is called using the -endarrow- as a delimiter. The word *delimit*, in programming, means *to fix the limits of* or *to separate*. Thus, the -endarrow- defines the limits of an arrow. The -endarrow- instruction informs the PLATO system that there are no more response judging commands associated with the previous -arrow- instruction. Its format is:

```
endarrow BLANK TAG
```

The -endarrow- instruction does not use a tag; the tag field remains blank. It is classified as a *response handling* instruction.

EXECUTION STATES

The instructions presented thus far are executed when the PLATO system is in one of two states: regular or judging. When the PLATO system is presenting information to the user, it is operating in the regular state. When the user's response is being evaluated, the PLATO system is switched to the judging state. Some PLATO instructions are only executed in the regular state and others only in the judging state. Thus, it is important for you to know which instructions are executed in each state.

Table 1 indicates the state in which each of the nine instructions presented in the previous section are executed.

TABLE 1
Execution States

Instruction	*State*	
	Regular	*Judging*
unit	X	
at	X	
write	X	
arrow	X	
answer		X
specs		X
wrong		X
no		X
endarrow	X	

The next activity explains the execution states in more detail and demonstrates how execution states affect your programming.

SUMMARY

The following chart summarizes the information presented in this activity.

Instruction	*Purpose*	*Type*	*Execution State*
unit	Controls lesson sequence	Lesson control	Regular
at	Specifies screen locations	Display	Regular
write	Displays text	Display	Regular
arrow	Requests student response	Response handling	Regular
answer	Judges responses as correct	Response handling	Judging
specs	Changes response judging options	Response handling	Judging
wrong	Judges specific answers as incorrect	Response handling	Judging
no	Judges unanticipated responses as incorrect	Response handling	Judging
endarrow	Separates (delimits) the processing of an arrow within a unit	Response handling	Regular

2-E. LESSON EXECUTION



Identify the effects of execution states on program execution.

It is sometimes difficult to visualize what a particular instruction will accomplish. For this reason, this CAI activity demonstrates the execution of the -unit-, -at-, -write-, -arrow-, -answer-, -specs-, -wrong-, -no-, and -endarrow- instructions. This tutorial demonstrates what occurs in a lesson when these instructions are specified.

CAI INTRODUCTION

The lesson is divided into eight parts:

a. Introduction
b. Using -at-, -write-, and -unit-
c. Getting and judging student responses (with the -arrow- instruction)
d. Judging with -answer- and -wrong-
e. Delimiting the arrow processing (with the -endarrow- instruction)
f. Allowing for unanticipated responses (with the -no- instruction)
g. Testing the PLATO system during the judging process
h. Modifying the judging process (with the -specs- instruction)

The lesson will take you through all eight parts; when you have finished the lesson, you will see an index that allows you to review any or all of the lesson parts. This lesson takes approximately thirty to forty-five minutes.

Now sign on to a PLATO terminal. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "Introduction: Using the PLATO Author Language," press LAB to take the lesson.

SUMMARY

The following paragraphs summarize only the new information presented in this activity. Other information in this activity is contained in the previous activity.

Displays

The specific location on the coarse grid at which the PLATO system begins to display the text specified by a -write- instruction is called *where*. It is set with the -at- instruction. The -at- instruction sets the left-hand margin for subsequent lines in a -write- instruction.

Judging

When the PLATO system encounters a judging instruction after an -arrow- has been executed, all processing stops and the PLATO system waits for a response. The student must indicate that the response is complete by pressing NEXT. Then the first and subsequent judging instructions following the -arrow- are executed until a match is found. When a match is found, the PLATO system switches from judging to regular state, and processes the regular instructions following the matched judging instruction and preceding the next judging instruction.

If the -no- instruction is not included, and no anticipated response is processed, the PLATO system writes *no* to the right of the unanticipated response and waits for a NEXT key press. When NEXT is pressed, the PLATO system erases the response and the *no* and waits for a new response. This can frustrate the student who doesn't know how to respond. The -no- instruction provides for unanticipated responses.

If a -no- instruction is included and an unanticipated response is entered, the PLATO system will not find a match, but will switch to a regular state to allow feedback specified in subsequent -write- instructions to be displayed. When NEXT is pressed, the *no* and the feedback are erased; the PLATO system waits for the next response.

The various -specs- instructions set specifications for the student's response. Their tags describe the specifications:

- *okextra* allows extra words if the key word is entered.
- *bumpshift* bumps all shifts from the student response. If bumpshift is used, no capitals can be included in the -answer- or -wrong- tags.
- *okspell* accepts recognized misspellings.
- *okorder* allows students to reverse words in a response that requires more than one word.

Specifications may be used in any combination if they are separated by commas.

2-F. HOW TO READ A FLOWCHART REPRESENTING STUDENT FLOW

Identify the flowcharting conventions and symbols used in *CREATE*.

Identify three general types of flowchart structures: linear, branching, and looping.

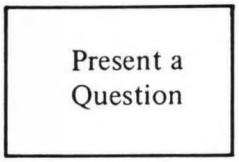
The person who builds a bridge is seldom the person who designed it. Yet the contractor and engineer involved in such a project require a concise means of communicating intent: they do it through blueprints. A similar need exists between a CAI developer and the person who programs the CAI activity. The "blueprint" used for communicating a CAI design is called a flowchart. The structures and conventions used in flowcharting are explained in this activity. In subsequent activities, you will be using flowcharts to code lessons.

DEFINITIONS OF TERMS AND SYMBOLS

A flowchart is a graphic means of showing a sequence of events which, in the case of CAI, is presentation of instruction. To accomplish this, a flowchart uses special symbols to represent the steps that occur in that process. These symbols, their meanings, and their uses differ from company to company and from discipline to discipline. Within *CREATE*, the following basic symbols are used consistently. They are the symbols basic to most data processing flowcharts and least likely to vary in meaning.

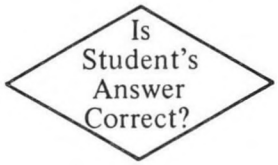
Processing, or Display

The rectangle represents a processing, or display, step. In a CAI design, it represents a point at which a student is given information or questions.



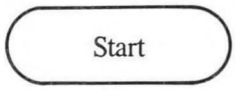
Decision

The diamond represents a decision point. Such a box in the sequence of events indicates a point at which alternate paths, or branches, occur. Which path a student is to follow depends on an identified variable. In the course of the activity, for example, there may be a point at which, based on their performance, students are branched either to a remedial activity or to advanced information. On a flowchart, this point would be represented by a diamond.



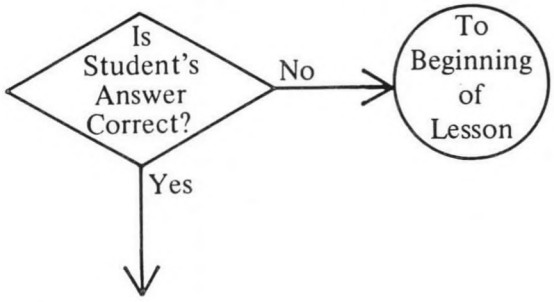
Terminal Point

The oval represents the beginning and end points of a flowchart's sequence of events.



Connector

A circle is a connector that represents an entry point from or exit point to another point on the flowchart. If, at a decision point in a lesson, students who do poorly are cycled back to the beginning, circles could be used to show where they leave the general flow and where they reenter it.



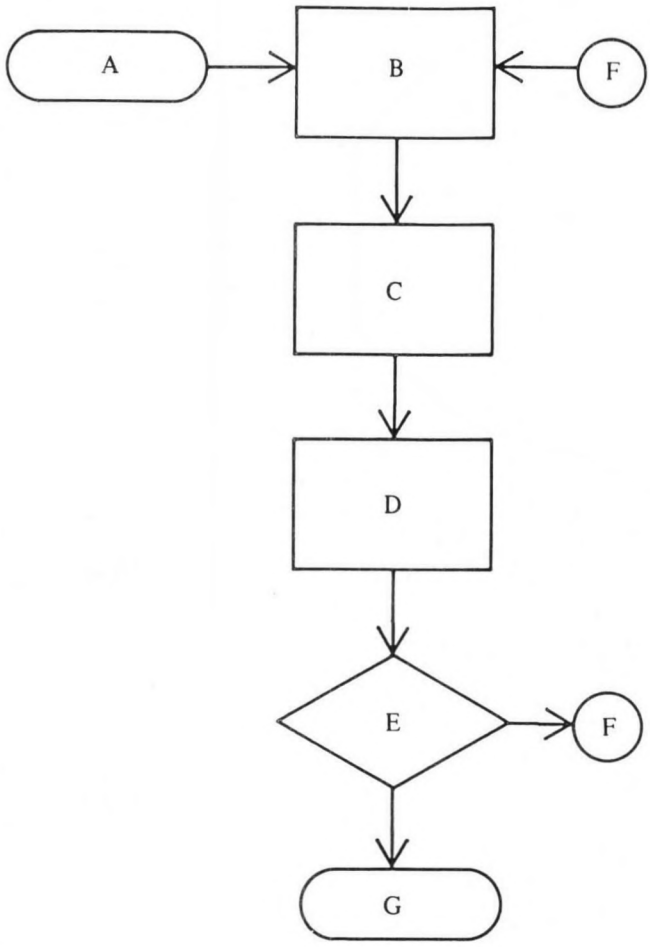
Flow

The arrow indicates a connection between the boxes, and the arrow point indicates the direction of the flow. Flowcharts are usually drawn in such a way that the progression of instruction begins at the top of the diagram and proceeds to the bottom.



CHECK YOUR UNDERSTANDING

Directions: Based on the flowchart shown, answer the following questions by writing the letters of the appropriate boxes in the spaces provided.



_____ 1. At which symbol does the flowchart begin?

_____ 2. At which symbol does the program end?

_____ 3. Which box represents a decision point?

_____ 4. A student who has gone from symbol *E* to symbol *F* will go to which symbol next?

_____ 5. Indicate a box that represents a processing step.

Answers

1. A 2. G 3. E 4. B 5. B, C, or D

A SAMPLE CAI FLOWCHART

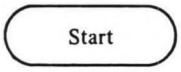
When developing a CAI activity, you must consider what the student and the program are doing at each step of the activity. Flowcharts are graphic means of describing these progressions. Consider the following instructional example:

> Welcome the students. Then present three problems involving the addition of fractions. If the students answer correctly, say "That's right!" If the students respond incorrectly, say "Too bad!" After the three problems, tell the students their scores and end the program.

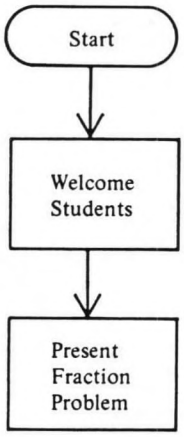
This simple CAI activity will be used to construct a sample CAI flowchart.

Linear Flow

Remember that every flowchart should start with the terminal point symbol. It is a good idea to write a brief description of the step within the symbol. So the flowchart begins like this:



The first steps are to welcome the students and present a problem. These are processing steps that should be presented by a rectangle. Each step has its own symbol. So far, the flowchart looks like this:



Notice two things about the flowchart:

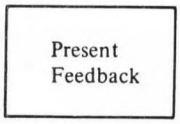
- There are separate boxes for the welcome and the problem because they are separate steps in the activity. (*Welcome* displays must not get mixed up in the fraction problems.)

- Arrows are used to illustrate the direction that the students, and thus, the computer, are to flow along this chart. The arrow always indicates the order of processing.

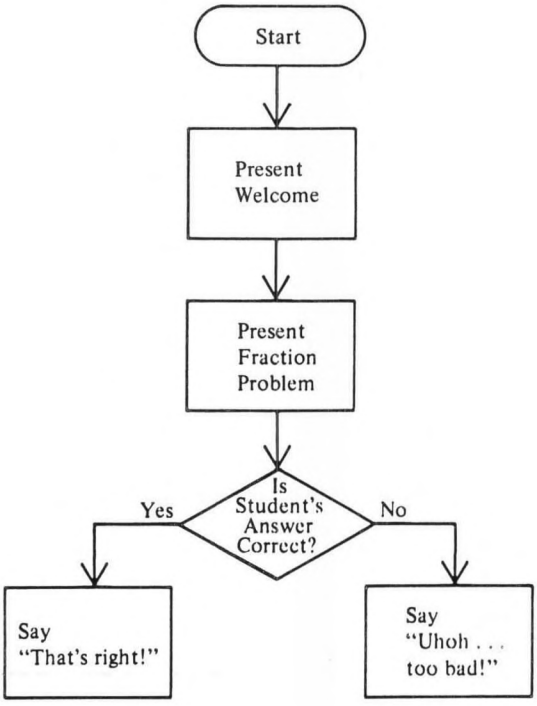
So far, the flowchart represents a linear flow.

Branching

Now another box could be added that says:



How do you know what feedback to present? This is obviously a decision point in the program, which should be represented by a diamond. The diamond is used because it has more than one point from which flow lines, or arrows, can emerge. This corresponds to the fact that decision points have alternatives. Decisions are usually based on *yes* or *no* and *true* or *false* conclusions. Decision points, however, can have more than two paths flowing from them. In the sample activity, there are two alternatives emerging from the decision box.



The two alternatives offered by the decision diamond are:

- *Yes* for situations in which the student's answer is correct
- *No* for situations in which the student's answer is incorrect

The student's path in the activity has *branched* at this point; that is, the path has deviated from the main, or linear, flow. It is possible that this flow from a branch point could become very complex. The student who answers correctly may be given additional problems; remedial information could be given to the student who answers incorrectly. Thus, it is these branching and decision points that provide the basis for much of the individualization in CAI activities.

For the sake of this example, these branches will merge immediately in order to present a second problem. The second and third problems are handled in the same way as the first. After all of the problems have been presented, a processing box indicates the score to the student. The flowchart for the finished activity is shown in figure 5.

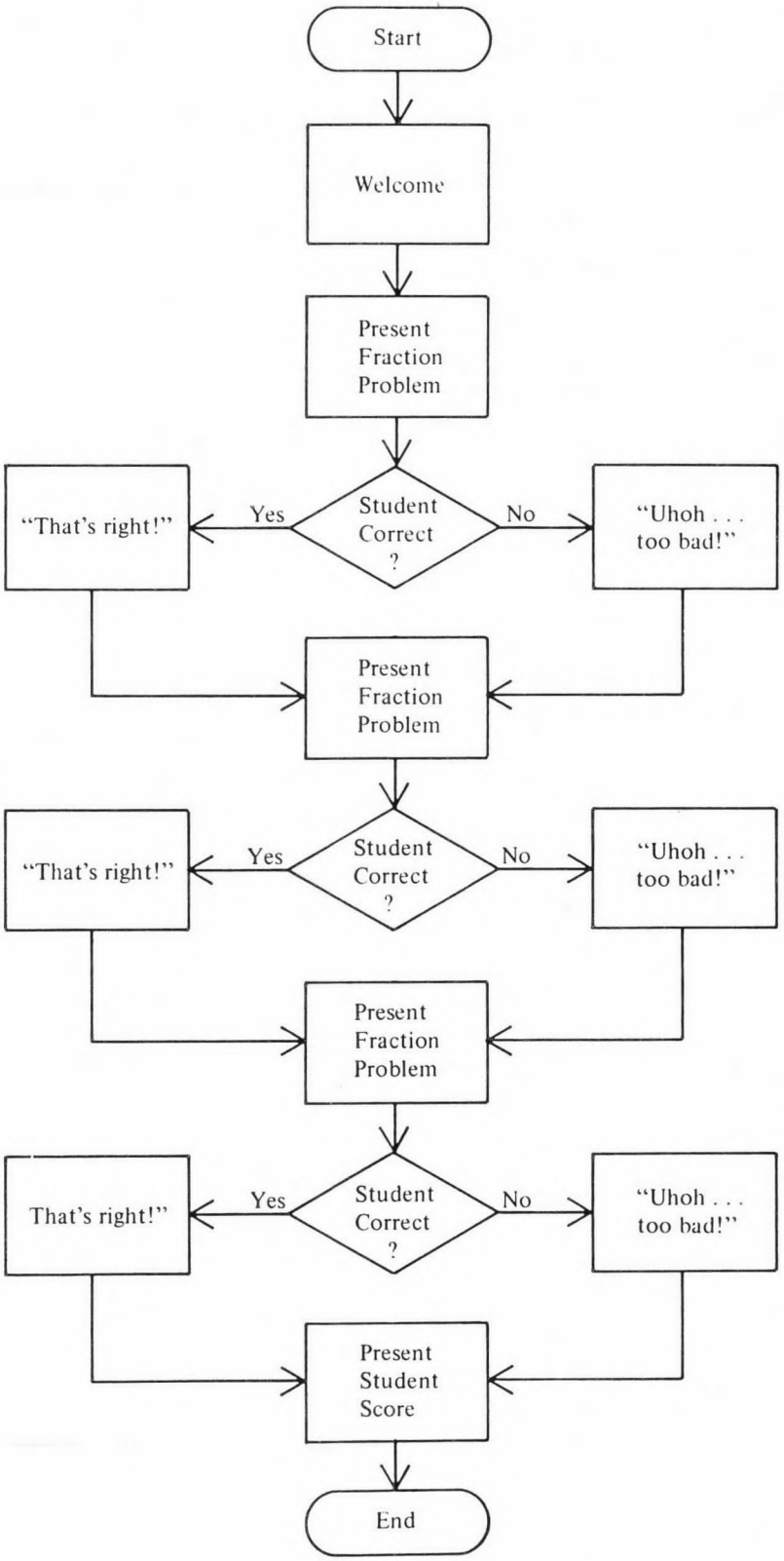
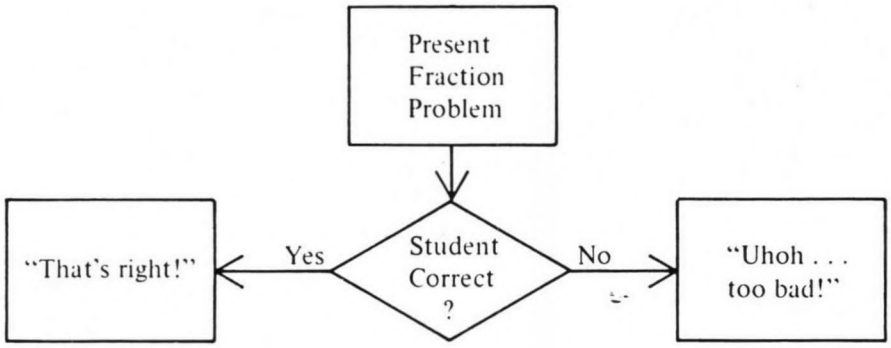


Figure 5. Flowchart showing branching program

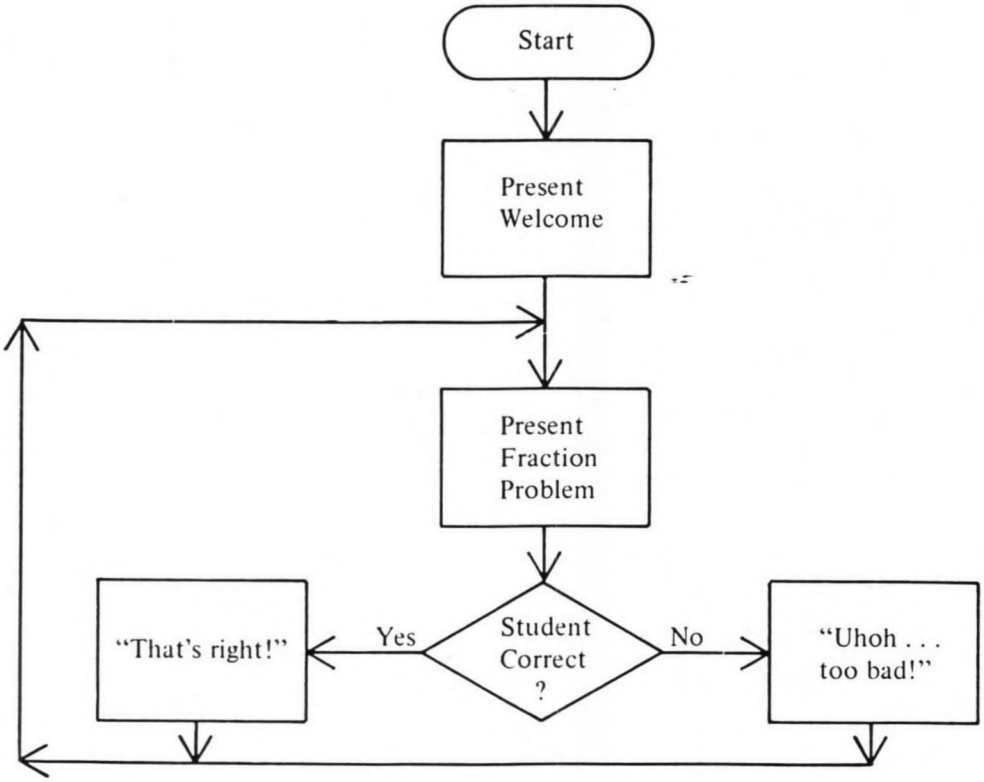
Looping

Now that the flowchart is finished, you can see that it is a finite process of steps that drill a student on adding fractions. Even a simple CAI activity can produce a detailed flowchart. A way in which to simplify such a flowchart is to employ a technique called *looping*.

Examine the flowchart and notice the pattern. This group of symbols occurs frequently:



Because of this recurrence, looping can be used. In looping, the flowchart goes through the same process a number of times and loops back on itself. Using looping results in a flowchart that looks like this:



In this new flowchart, the student still receives a welcome page, a problem, feedback, another problem, and so forth. The trouble with this arrangement, though is that students who fail repeatedly will continue to receive problems–there is no end to the loop. This is called an *endless loop*. It can be solved by adding a new decision box to the flowchart. If the student has received three problems, the program tallies the score and ends. Figure 6 shows the use of looping.

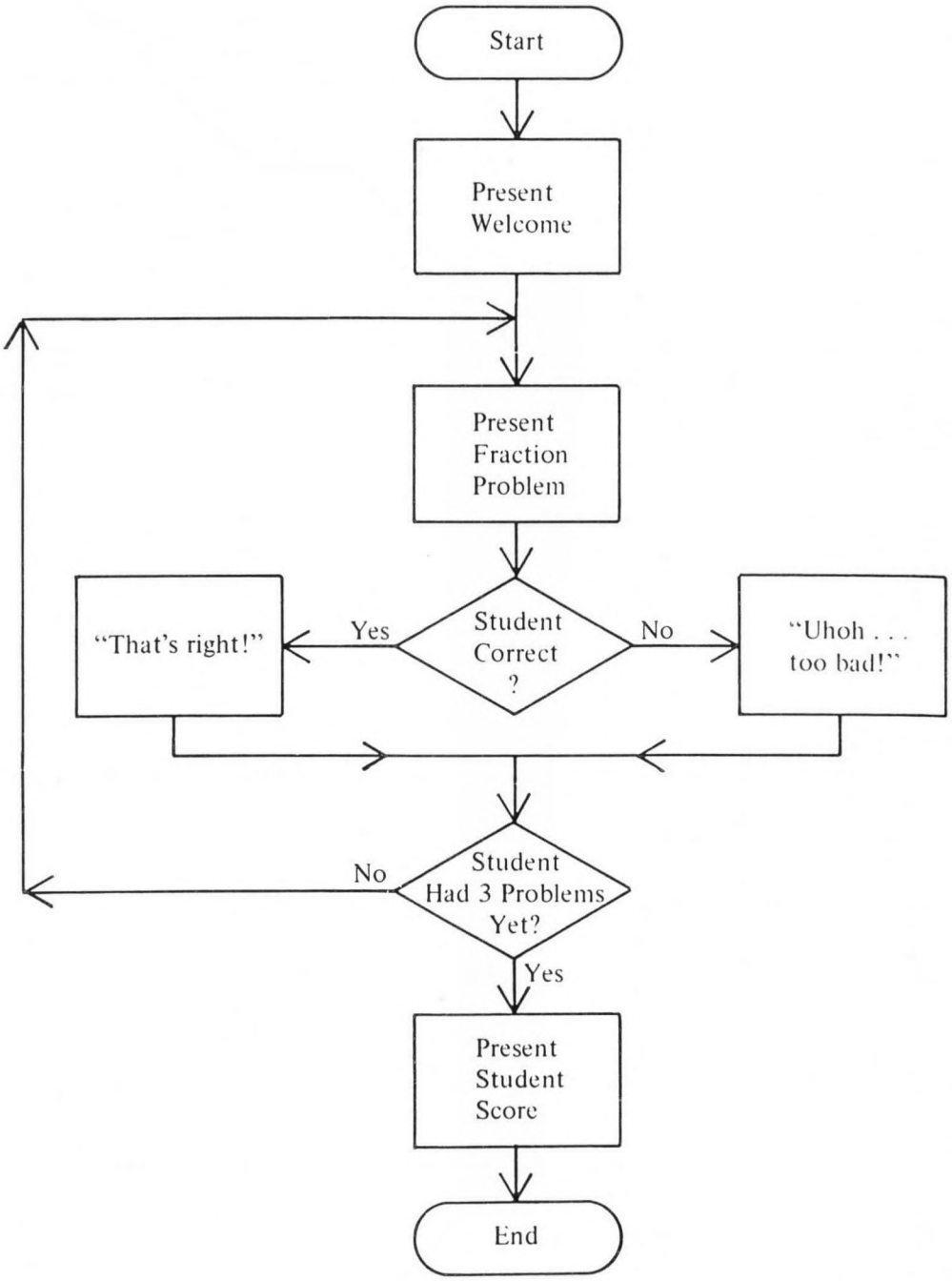
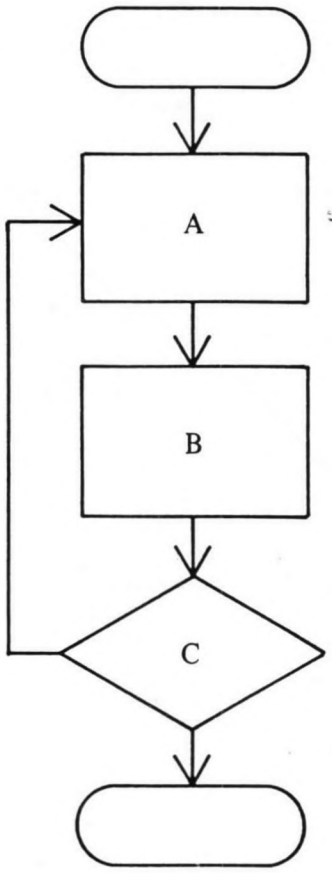


Figure 6. Flowchart showing looping program

Notice that a loop must be followed by a decision point, or the student is not able to exit from the loop. In a sense, the loop is simply one alternative to a branch, but it is an alternative that directs learners to sections of the lesson they've already gone through. On complex flowcharts, connector symbols may be needed to indicate looping.

CHECK YOUR UNDERSTANDING

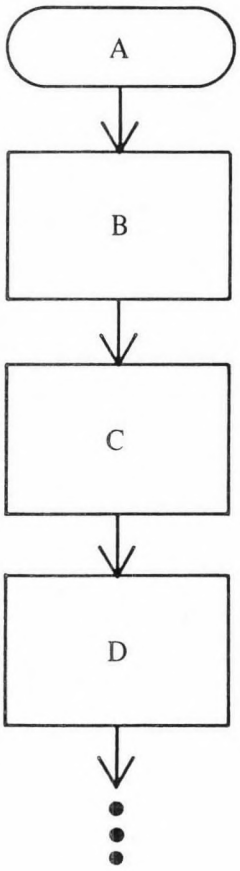
Directions: For each of the following diagrams, indicate whether a linear, branching, or looping structure is represented. Briefly explain your answers.



1. _____

_____

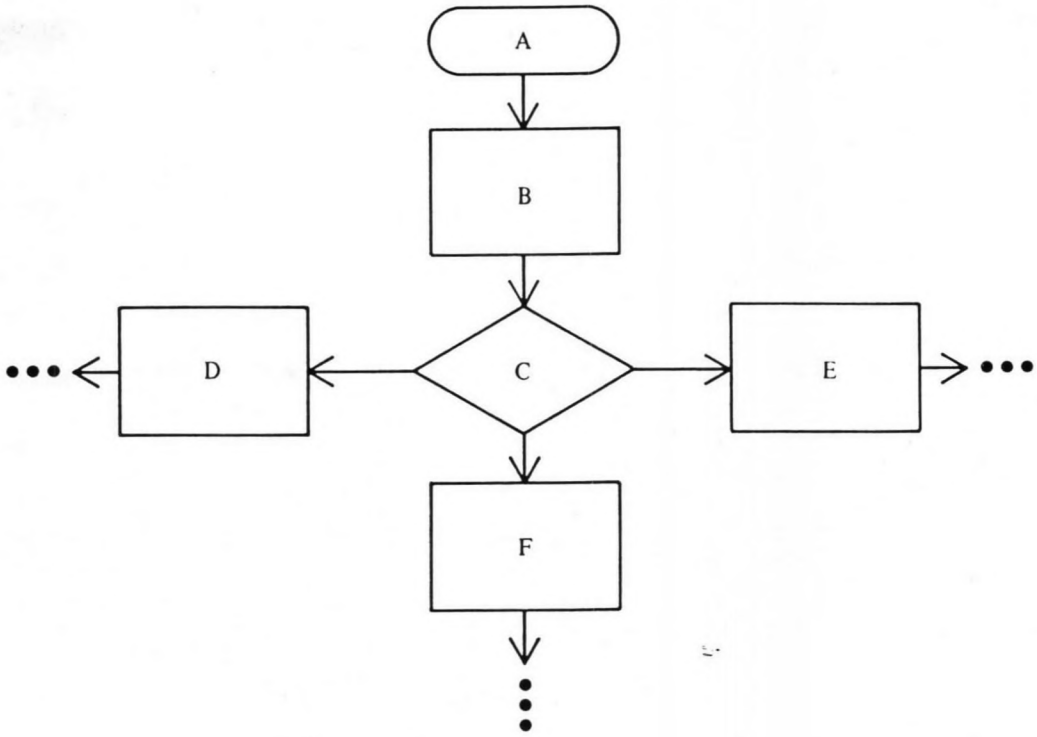
_____



2. _____

_____

_____



3. _____

_____

_____

Answers

1. Looping. Notice that box *C* is a decision point, and that one of the alternatives directs students to sections of the flowchart they've already been through (boxes *A* and *B*).
2. Linear. Students proceed straight through a series of steps.
3. Branching. Again, box *C* is a decision point, from which three alternatives branch.

2-G. INTERACTIVE PROGRAM



Identify correct and incorrect instruction sequence.

At this point, you know the purpose and format of nine PLATO Author Language instructions. Instructions must, however, be sequenced in certain ways to allow the program to execute as desired. This activity allows you to write a program and to experiment with instruction sequences in order to determine sequences that will produce the desired results.

PROGRAM DIRECTIONS

The following pages contain a flowchart and simple coding directions similar to what you might receive from a CAI developer. Study these pages, and write lesson code to produce the desired results. The nine instructions you learned in this unit are sufficient for this exercise.

CODING DIRECTIONS

The boxed information contains the questions to be asked and the feedback to be given for student responses.

1. Red and yellow are two primary pigments. What is the third?

Student response	*Is response correct?*	*Feedback for response*
blue	yes	Good!
green	no	Green is a primary color of light, not pigment.
purple or violet	no	That's a result of mixing primary pigments.
orange	no	That's a result of mixing yellow and red.
all other answers	no	Choose from: orange, green, blue, or purple.

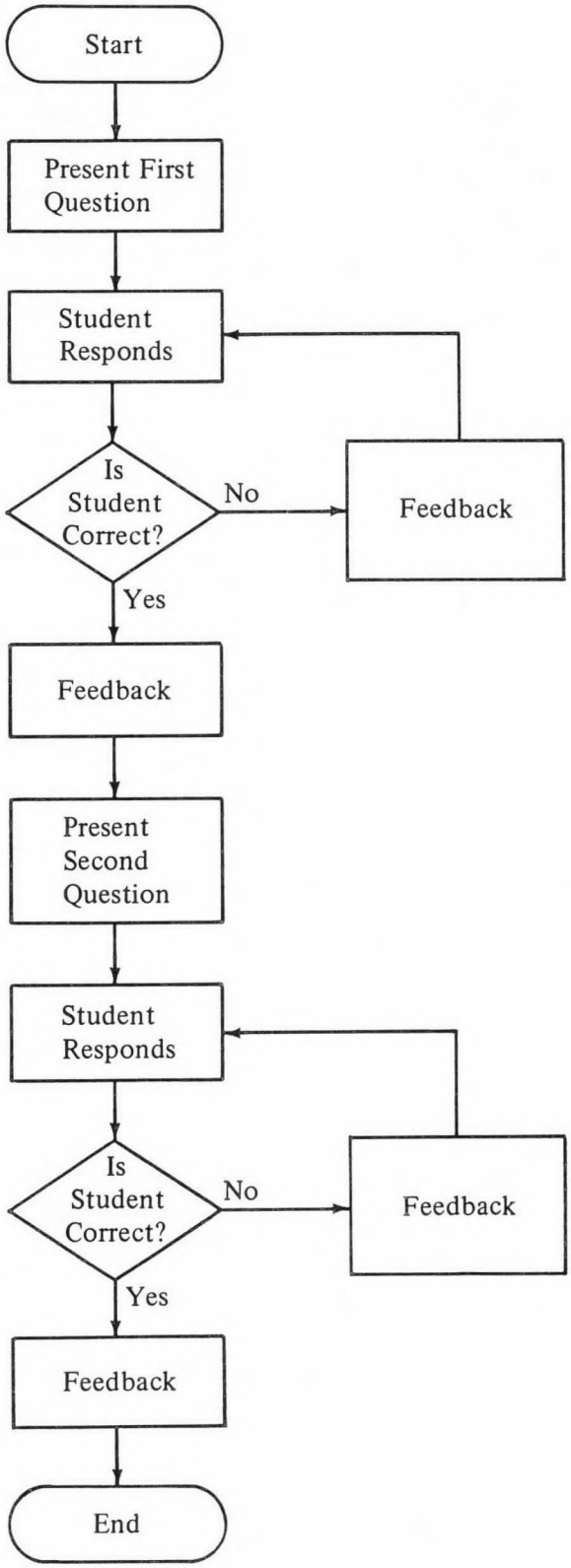
2. Secondary colors result from mixtures of any two of these three primary pigments. What color is the result when red and yellow are mixed?

Student response	*Is response correct?*	*Feedback for response*
orange	yes	ok
purple or violet	no	That's blue and red.
green	no	BLUE and yellow!

Response instructions: Capitalization or minor misspellings are acceptable. The student should be required to answer correctly before proceeding.

Additional: It would be nice if the student could see the first question while answering the second. This is not required, however.

FLOWCHART


Start
Present First Question
Student Responds
Is Student Correct?
No
Feedback
Yes
Feedback
Present Second Question
Student Responds
Is Student Correct?
No
Feedback
Yes
Feedback
End

CAI INTRODUCTION

When you are satisfied with your code, sign on to a Control Data PLATO terminal. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "Making the PLATO Author Language Work," press LAB to take the lesson.

This lesson will ask you to identify and correct errors in code meant to satisfy the requirements specified in the coding directions. Use your own completed code to help identify the errors. When you have completed the CAI lesson, correct your code, if necessary. Sample code is provided on the next page.

SUMMARY

The following sequence of instructions will satisfactorily carry out the actions called for in the coding directions.

```
unit     primary  $$any acceptable unit name
at       510
write    Red and yellow are two primary pigments.
         What is the third?
arrow    810
specs    bumpshift,okspell
answer   blue
write    Good!
wrong    green
write    Green is a primary color of light, not pigment.
wrong    (purple,violet)
write    That's a result of mixing primary pigments.
wrong    orange
write    That's a result of mixing yellow and red.
no
write    Choose from: orange, green, blue, or purple.
endarrow
at       2110  $$1810,1910,2010 also acceptable
write    Secondary colors result from mixtures of any two
         of these three primary pigments. What color is
         the result when red and yellow are mixed?
arrow    2510
specs    bumpshift,okspell
answer   orange
wrong    (purple,violet)
write    That's blue and red.
wrong    green
write    BLUE and yellow!
endarrow
```

3

Entering and Editing Code

This unit explains how to enter and edit code using the PLATO system. It discusses the sign-on procedure, editing directives, and search directives. It also provides you with the opportunity to enter and edit the code you wrote in unit 2.

INSTRUCTIONAL OBJECTIVES

At the end of this unit, you should be able to:

- Identify function keys used to move between major PLATO displays
- Sign on to the PLATO system as an author
- Recognize the format of lesson code
- Identify steps that must be taken to inspect code
- Identify the effects of the *i*, *d*, *f*, and *b* editing directives
- Identify steps taken to create a block
- Identify procedures for using the PLATO system to discover and eliminate condense errors
- Identify the effects of the $\times$, $\div$, +, Σ, –, Δ, *COPY*, *COPY1*, *r*, *s*, *is*, and *out* directives
- Enter and edit code using the PLATO editor
- Identify ways lesson structure affects programming
- Identify ways in which ECS affects programming
- Identify the effects of the *u*, *c*, and *x* search directives
- Identify procedures for receiving help on editing

LEARNING ACTIVITIES

_____ 3-A. Control Data PLATO Displays. CAI: This activity discusses the author sign-on procedure.

_____ 3-B. Author Sign-on. Text/CAI: In this activity, you will sign on to the PLATO system as an author and enter information in the lesson information display.

_____ 3-C. A First Look at Code. Text/CAI: This activity gives you the opportunity to inspect the code for a CAI lesson you completed in unit 2.

_____ 3-D. Text Editing, Lesson One. Text/CAI/Exercise: This activity explains the use of the *i*, *d*, *f*, and *b* editing directives.

_____ 3-E. Condense Errors. Text/CAI: This activity allows you to create a block and use the PLATO system to discover and eliminate condense errors.

_____ 3-F. Text Editing, Lesson Two. Text/CAI/Exercise: This activity discusses and allows you to practice the X, ÷, +, –, Σ, Δ, *COPY*, *COPY1*, *r*, *s*, *is*, and *out* editing directives.

_____ 3-G. Entering Your Own Code. Text/CAI: In this activity, you will enter and edit code using the PLATO editor.

_____ 3-H. Lesson Space Structure and Extended Core Storage. Text: This activity explains the structure of a lesson space and ways that extended core storage affects programming.

_____ 3-I. Text Editing, Lesson Three. CAI: This activity discusses the use of the *u*, *c*, and *x* search directives.

_____ 3-J. Editing Help. Text/CAI: In this activity, you will learn to access a help sequence that can help you while you are editing.

TESTING DIRECTIONS

Now sign on to a Control Data PLATO terminal. If you are very familiar with the subject of entering and editing code, you may wish to test out some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.

3-A. CONTROL DATA PLATO DISPLAYS

Identify function keys used to move between major PLATO displays.

For previous PLATO activities, your student sign-on has taken you to a CAI prescription page, from which you have accessed CAI lessons. For an author, however, a different sign-on procedure is required.

CAI INTRODUCTION

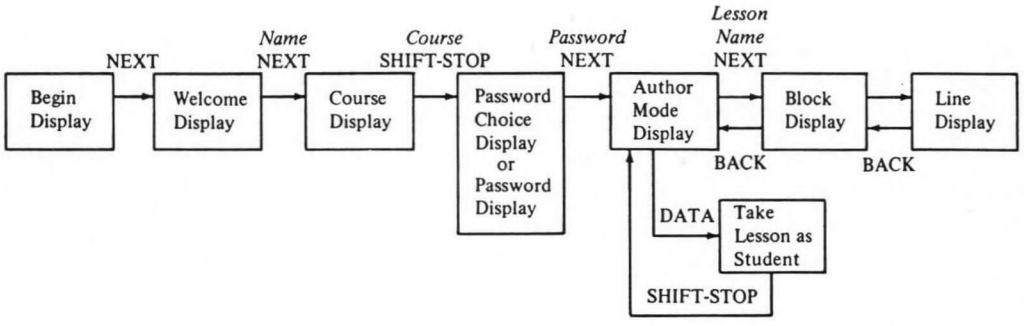
This activity discusses the author sign-on procedure. The activity has three parts:

- "PLATO Display Vocabulary." This part will help you recognize terms for important Control Data PLATO displays.
- "PLATO Navigation," This part tells you how to move between major PLATO displays.
- "The PLATO Display Regatta." This part allows you to practice moving between major displays on your own.

Now sign on to a Control Data PLATO terminal using your *student* sign-on. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "PLATO Landmarks," press LAB to take the lesson. When you have completed this activity, return to your text to read the summary.

SUMMARY

The following chart summarizes the major displays and the actions you should perform to move through these displays.



3-B. AUTHOR SIGN-ON



Sign on to the PLATO system as an author.

The last activity explained the author sign-on procedure. In this activity, you will actually sign on as an author for the first time. You will also enter information in the lesson information display.

Before going to the PLATO terminal, you will need the following information:

Author sign-on name: _____

Group name: _____

File name: _____

Contact the learning center technician to obtain this information. Record it in the space provided for future reference.

SIGN-ON

Now that you have the necessary information, go to a PLATO terminal and:

- On the welcome display, enter the author sign-on name you were given and press NEXT.
- On the course display, enter the group name you were given and press SHIFT-STOP.
- You will now be asked to invent a password. Choose any combination of up to ten letters or numbers that you can easily remember. Then press NEXT.

AUTHOR MODE DISPLAY

You are now on the author mode display. From this display, you can look at the *code* for most Control Data PLATO lessons by typing the lesson name and pressing NEXT, and by entering the required security code, if known. You can *execute* most lessons by typing the lesson name and pressing DATA. The lesson executes as it does in the student mode.

AUTHOR INFORMATION DISPLAY

From the author mode display, type your file name and press NEXT. You will now see the block display. Press the DATA key to get to the directory index. Type the letter *a* to see the author information display. An example of this display is shown in figure 7.

```
Lesson name ---- judy

Account -------- codev


Press the associated number to change an entry.

Author Information:
     1. Name ---------------
     2. Dept./Affiliation --
     3. Telephone number ---

Lesson Information:
     4. Subject Matter -----
     5. Intended Audience --
     6. One line description ----------
```

Figure 7. Author information display

Fill in the author information, particularly the one-line description. At this time, because you do not know what lesson you will be programming, type in *work space*. The system will return you to this display until the one-line description is entered.

From the author information display, press NEXT to see the associated files display. Do not be concerned with this display at this time; it deals with more advanced programming techniques than will be taught in this PLATO Author Language course.

CODEWORDS DISPLAY

From the associated files display, press NEXT to see the codewords display. An example of this display is shown in figure 8.

```
Press the associated number to change an entry.

SECURITY CODES:
     1. To change lesson ----  blank--OPEN TO ALL
     2. To inspect lesson ---  blank--OPEN TO ALL
     3. To access common ----  blank--OPEN TO ALL
     4. To -use- lesson -----  blank--OPEN TO ALL
     5. To -jumpout- to -----  blank--OPEN TO ALL
     6. To -attach- a file --  blank--OPEN TO ALL

Access to file by system personnel:
     7. System Access -------  ALLOWED
```

Figure 8. Codewords display

The codewords display lists security information. To prevent your lesson from being changed by another author, make up and enter a code in the *to change lesson* space. If you want to prevent others from inspecting your lesson, you must also enter a code in the *to inspect lesson* space. Leaving this space blank allows access to all.

The *to change lesson* security code may be different from your password. It is wise to set this security code immediately upon entering a new lesson space. This assures that no one will be able to edit your code without your permission.

If you forget your security change code, inform the learning center technician. The forgotten change code cannot be recovered, but the learning center technician can replace it with a new change code.

If you set the *to inspect lesson* security code, you may wish to choose a different code. Then you will be able to allow others to see your code by giving them the inspect code, but they will not be able to edit it because they do not have the change code.

Press BACK until you return to the author mode display, and SHIFT-STOP to sign off. Do not leave the terminal until you see the begin display (Press NEXT to begin).

If you had pressed NEXT from the codewords display, you would have been taken to the editing specifications display. Do not be concerned with this page at this time. You can use the above instructions for signing off from any of the displays discussed in this activity.

SUMMARY

To sign on as an author, you need an author sign-on name, a group name, and a file name. To sign on, go to the terminal and, on the welcome display, enter your sign-on name. Press NEXT. Enter your group name. Press SHIFT-STOP. Enter your password. Press NEXT.

At the author mode display, you can look at lesson codes by typing the file name, pressing NEXT and entering the required security codes, or you can execute a lesson by typing the lesson name and pressing DATA.

At the author mode display, when you wish to program a lesson, type your lesson space name and press NEXT. Fill in the author information display and the codewords display.

3-C. A FIRST LOOK AT CODE



Recognize the format of lesson code. **Identify steps that must be taken to inspect code.**

Thus far, this module has included a great deal of discussion on coding a Control Data PLATO lesson. You have not, however, seen any actual code. This activity gives you the opportunity to inspect the code for the CAI lesson you completed in unit 2.

CAI INTRODUCTION

In this short activity you will sign on as an author to look at a sample of code. Notice how the code looks on the screen. In later activities, the code you write will appear on the screen in a similar format. To complete this activity:

1. Go to the PLATO terminal. Enter your name, your group name, and your password.
2. When you see the author mode display, type in *Øplato3* and press NEXT. Now you should see this display:

```
Press LAB to inspect "Øplato3"

or

Press DATA to try this lesson.


Or, to edit it:

Type the security code of this lesson, then press NEXT:
```

   This lesson has no inspect security code, so you will be allowed to inspect the code.

3. Press LAB to inspect this lesson. Notice that the words INSPECT ONLY appear at the top of the next display. This means that no matter what keys you press, you cannot alter the code.

4. Press the letter *a* to see the lesson security/use information. Notice that option *a. To change lesson* has been filled in. You can't change the lesson code.
5. Press NEXT to see the author information display.
6. Press BACK to go back to the block display.
7. Press *b* to see the sample code.
8. When you see the sample code, press the space bar twice to see all available code. Study the format of the code.
9. When you have finished looking at the code, press BACK to get to the block display. Press BACK again to get to the author mode display. Then press SHIFT-STOP to sign off the terminal.

3-D. TEXT EDITING, LESSON ONE



Identify the effects of the *i*, *d*, *f*, and *b* editing directives.

Unit 2 presents enough instructions to allow you to code a simple lesson. You are now ready to learn to enter code into the Control Data PLATO system. To enter code, you will use the Control Data PLATO editor, a powerful device that allows you to enter and revise code.

CAI INTRODUCTION

This activity explains the purpose of the editor and four basic editing directives:

- *insert* (i), which allows you to insert lines of code
- *delete* (d), which allows you to delete lines of code
- *forward* (f), which allows you to move lines of code forward
- *backward* (b), which allows you to move lines of code backward

Now sign on to a Control Data PLATO terminal using your *student* sign-on. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "The Elements of PLATO Text Editing," press LAB to take the lesson. When you have completed this activity, return to the text summary.

SUMMARY

The editor saves the code you write. It allows you to enter, revise, and read your code for notes and lessons. The editor provides a clue to how much space is left for writing. This clue appears in the upper left-hand corner of the screen. The arrow in the upper right-hand corner waits for an indication of the type of editing you wish to do. If, for example, you wish to insert a line after line 2, *i2* appears next to the arrow, and the screen displays the words INSERT MODE. Another arrow indicates where you can insert a line. To go on to the next line, press the NEXT key.

The *insert* directive (*i* followed by a number and a depression of the NEXT key) allows you to add lines of text. The directive *i3* will allow you to insert a line after the third line of code on the screen. A BACK key press ends insert mode.

The *delete* directive (*d* followed by a number and a depression of SHIFT-HELP) allows you to delete lines at the top of the screen. The directive *d1* deletes the first line; the directive *d4* deletes the top four lines. When deleting, watch for a change in the space indication at the top of the screen to confirm deletion.

The *forward* directive (*f* followed by a number and a depression of the NEXT key) moves the frame of reference forward. It moves the top line on the screen closer to the end of the text.

The *backward* directive (*b* followed by a number and a depression of the NEXT key) moves the frame of reference back. To see the beginning of a sequence of text, the frame of reference must be as far back as it can go. The directive *b100* can be used to be sure you back up enough.

The editor that is available for the PLATO system has special instructions. Press the HELP key to receive specific instructions while using the editor.

CHECK YOUR UNDERSTANDING

Section 1 (*f, b*)

Directions: To complete this section of the exercise:

1. Sign on to a PLATO terminal as an author.
2. At the author mode display, type *∅plato3* and press NEXT.
3. Press LAB to inspect the lesson.
4. At the block display, press *b* to see the sample code.
5. Press the space bar three times to see all code in the sample.
6. Press *b100* and the NEXT key to return to the beginning of the code.
7. Press the + key several times. (After each press, notice that your code has advanced one line. A different line appears at the top of the screen with each press.)
8. Experiment with the *f* and *b* directives until you are proficient in their use.
9. When you have finished practicing, press BACK until you see the author mode display. Press SHIFT-STOP to get back to the *Press NEXT to begin* display. Then go on to section 2 of this exercise.

Section 2 (*d, i*)

Directions: Answer the following questions in the blanks provided.

1. What directive would you use to delete the top three lines of your code?

   _____

2. What directive would you use to delete the top ten lines of your code?

   _____

3. What should you check to confirm that a line has been deleted?

   _____

4. If you want to insert a line after line 3 on the display, what should you press?

   _____

5. If you want to insert a line after line 5 on the display, what should you press?

   _____

6. If 13 lines of code are visible on the screen, what should you do to delete lines 1Ø and 11?

   _____

7. If 25 lines are visible, what should you do to delete lines 17, 18, and 19?

   _____

8. What do you press to end the insert mode?

   _____

Answers

1. Press *d3*, SHIFT-HELP
2. Press *d1Ø*, SHIFT-HELP
3. The space indicator
4. Press *i3*, NEXT
5. Press *i5*, NEXT
6. Press *f9* to bring line 1Ø to the top of the screen, press *d2*, press SHIFT-HELP, check the space indicator
7. Press *f16* to bring line 17 to the top, press *d3*, press SHIFT-HELP, check the space indicator
8. Press BACK



3-E. CONDENSE ERRORS

Identify steps taken to create a block.

Identify procedures for using the PLATO system to discover and eliminate condense errors.

Some coding errors can prevent the PLATO system from immediately condensing and, consequently, from executing your lessons. Thus, it is important that you be able to identify and correct these errors.

CAI INTRODUCTION

Because it is impossible to describe each of the errors that can be made while coding a lesson, this activity explains how the PLATO system can help you identify and correct errors in your code. You will learn to create a block to enter code, and to use the PLATO system to discover condense errors.

Creating a Block

To complete this activity, you must first learn how to create a block. You cannot enter code until you have done so. To create a block, follow these steps:

1. Sign on to a Control Data PLATO terminal.
2. When you see the author mode display, type in your file name and press the NEXT key. If, for some reason, you are asked for your security code, type in the lesson security code, not your password. You should now see the block display. This display should list only one block (*a. block directory*). You must now create a block. Type in the *capital* letter of the block that precedes the block you wish to create. (Lower case letters will take you into the block.) In this case, type the letter *A*.
3. You should now see the create-a-block display:

```
Press...

    NEXT  for a normal TUTOR block
    LAB   for copy-a-block
>
      1   for a common block
      2   for a charset block
```

```
3    for a micro block
4    for a leslist block
5    for a vocabulary block
6    for a lineset block
7    for a listing block
8    for a text block
```

This display gives you the options for block creation. For now, ignore the information associated with the LAB key and all the numbers. Simply press NEXT to create what is called a *normal TUTOR block.*

4. You will now be asked to name the block you are creating. Type in *test* as the block name. Press NEXT. You are now ready to enter code.
5. Press the BACK key; it takes you back to the block display. Notice that your block is gone; this is because no code was entered in your block.
6. Create a block again. Try to do this without looking at your text. If you can't remember the procedure, repeat steps 2 through 4.
7. When you see the first blank display page, type an *i* and press the NEXT key. You are now in insert mode.
8. Now insert an asterisk (shifted 8) on the first blank display page of your block. Press BACK to return to the block display. Your block remains because it now contains code. Notice the two asterisks in front of block *test*. The asterisks tell you that *test* was the last block you were in before you went to the block display.
9. Now reenter block *test* (press *b*).
10. Delete the asterisk (press *d1* and SHIFT-HELP). You are now ready to enter code and learn about condense errors.

Condense Errors

In this portion of the exercise, you will enter erroneous code to see how the PLATO system can help you eliminate (debug) coding errors. To complete this section, follow these steps:

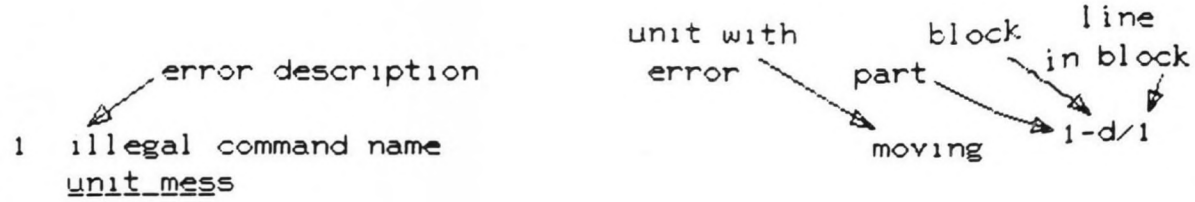
1. Go into insert mode by typing the letter *i* and pressing NEXT (*i* and *il* both set the editor at the first line of the display).
2. Type in this instruction: *unit mess*. Leave only one space between the command and the tag.
3. Press SHIFT-STOP to condense this instruction. You should now see this display explaining the error in the code you entered:

```
Errors found when condensing lesson 'Øplato3'
Errors: 1      Units: 3      Condensed Words: 248
```

```
1  illegal command name                          moving    1-d/1
   unit_mess
```

```
More info on which error? >                      HELP is available
SHIFT-NEXT to edit error                         DATA to try lesson
```

4. Press the number 1 and the NEXT key to receive more information on the error. You will see this display:




```
The name in the command field (1st 8 characters
of the line) is not recognized as a TUTOR command.
```

```
For detailed information on another error,
enter its number and press NEXT : >

Or press:
   NEXT or BACK to return to main display
   SHIFT-NEXT to edit this error
```

5. To be selective when editing errors, enter the number of the error you wish to edit and press SHIFT-NEXT. The error you wish to edit will appear at the top of the screen.
6. Once you have received the condense error message and reentered your block in which the error occurred, you may use the *show error* option to review the condense error message without recondensing your code. To do this, type *se* and press NEXT.
7. Return to edit the error by pressing SHIFT-NEXT.
8. The error in your code involves a spacing problem. To correct this error, first delete the existing line (type *d1* and press SHIFT-HELP).

9. Press *i* and the NEXT key. Now type in *unit*, press the TAB key, and type in *mess*.
10. Press SHIFT-STOP to condense the lesson. A blank screen should appear. You are now in the lesson, but because you haven't entered any display instructions, no text appears.
11. Now press SHIFT-STOP. You should be back on the author mode display.
12. Press NEXT. Note that you are still in your lesson; the PLATO system "remembers" it to save time in reentering your block. When you SHIFT-STOP out of a lesson without signing off the terminal, you can always return to it in this manner unless you have entered a special instruction in your lesson to prevent this occurrence, or unless you have a condense error in the lesson.
13. Press *b* to return to your code. Enter insert mode. Now type: *at 505*. Be sure to press the TAB key after entering the command and before entering the tag.
14. Press NEXT to move down a line.
15. Type: *write hi* (use the TAB key).
16. Press SHIFT-STOP to condense this code. You should now see the tag of the -write- instruction as it would appear to a student.
17. Press SHIFT-STOP to get out of your lesson. At the author mode display, press NEXT to reenter the lesson. Press *b* to return to your code. Delete the three lines of code.

Additional Practice

You should now be able to:

- Enter code
- Condense code
- Request information on code errors
- Edit code

Using the directives you know (*i*, *d*, *f*, *b*, *se*), practice using the editor. You may wish to enter and condense the following instructions to see a variety of condense errors. You may not understand all of the PLATO system's explanations; however, you should understand enough to identify the errors. (After seeing the condense errors for an example, be sure to delete the instructions you have entered in order to experiment further.)

1. Simply type in *at* with no tag and condense with SHIFT-STOP. In this case, the command *at* must have a tag. The PLATO system will discuss this error.

2. Type in *at 5Ø*. (Press the TAB key between the command and the tag.) The PLATO system will tell you that you must specify a coarse-grid position, such as Ø15Ø. In this case, the *specific* error (5Ø) is not mentioned in the PLATO system's instruction, but the basic problem is explained.

3. Type:

```
unit     letsmakeamess
at       ØØ5Ø
write mess
```

In this case, your program will have three errors (length of unit tag, lack of line specification in -at- tag, and inappropriate command). For the PLATO system's explanation of each error, press the number of the error you wish to have explained.

IMPORTANT NOTE

When you have finished practicing, delete all of your lines of experimental code. This will leave your lesson file blank. You will need this space in a later activity. When you have deleted all of the lines of code, press BACK. Your block should be gone from the block display. The next time you enter your file, you will create a new block.

Press BACK to go to author mode display, then press SHIFT-STOP until the screen displays the message *Press NEXT to begin*. Then go on to the next activity.

3-F. TEXT EDITING, LESSON TWO



Identify the effects of the ×, ÷, +, Σ, –, Δ, COPY, COPY1, *r*, *s*, *is*, and *out* directives.

While it is possible to enter and edit code with the *i*, *d*, *f*, and *b* directives, editing can be accomplished much faster using the additional directives presented in this CAI lesson.

CAI INTRODUCTION

This lesson explains how to use the following keys to move code forward and backward:

× (times) key
÷ key
+ key
Σ (shifted +) key
– key
Δ (shifted –) key
COPY key
COPY1 (shifted COPY) key

It also explains how to use the *r*, *s*, *is*, and *out* directives to replace and save lines, insert lines, and restore blocks of code when errors have been made.

Now sign on to a Control Data PLATO terminal using your *student* sign-on. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "Editing Time-Savers 1: More Editing Directives," press LAB to take the lesson. When you have completed this lesson, return to this activity to read the additional notes and the summary.

ADDITIONAL NOTES

Pressing Σ (shifted +) brings the last line of code in a block to the top of the display. It produces the same results as typing *f* and a large number, such as *f1ØØ*. Pressing Δ (shifted –) brings the first line of code in a block to the top of the display. It produces the same results as typing *b1ØØ*. Σ and Δ are faster, however, and will make sure that you move all the way forward or all the way back in your block of code.

SUMMARY

The following editing directives make editing faster and easier:

- The × (times) key moves the display forward one frame.
- The ÷ key moves the display backward a few lines.
- The + key moves the display forward one line.
- The – key moves the display back one line.
- The Σ (shifted +) symbol moves the display to the end of a block of code.
- The Δ (shifted –) symbol moves the display to the beginning of a block of code.
- The COPY key will copy one word per press to allow you to edit specific words.
- COPY1 (shifted copy) will copy the remainder of the original line.

The *r* (replace) directive allows you to replace a line of code.

The *s* (save) directive saves the number of lines you specify at the top of the screen in a save buffer. For example, *s2* saves the top two lines. This directive allows you to move material from one place to another within a block, between blocks, or between lessons.

The *is* (insert save) directive allows you to insert the lines you have saved after the *n*th line in your display. For example, *is1* inserts at the top of the screen, while *is5* inserts after the fifth line.

The *out* directive will restore a block of code to the state it was in when you first accessed the block. It is often used to correct deletion mistakes. If you use the *out* directive in a block that was empty when you began, it will delete any code you have entered.

Δ and Σ function the same as f1ØØ and b1ØØ, but are faster, and are sure to bring the code all the way forward or all the way back.

CHECK YOUR UNDERSTANDING

Proficiency in the use of editing directives takes practice. This CAI activity allows you to practice the editing directives presented in activity 3-G. The practice is divided into two sections:

- Section 1 (×, ÷, +, –, Σ, Δ)
- Section 2 (*r*, *s*, *is*, *out*)

Now go to the PLATO terminal to begin this exercise.

Section 1 (X, ÷, +, –, Σ, Δ)

1. Sign on as an author. When you see the author mode display, type *∅plato3*.
2. Press LAB to inspect the lesson. At the block display, type *c* for exercise two.
3. You should now see seven lines of code on the screen. Press the X key. *This is line 7* should appear at the top of your screen, because it was the bottom line of the previous display.
4. Press the space bar. You should now see nineteen lines of code.
5. Press the X key. *This is line 25* should appear at the top of the display. It was the bottom line of the previous display.
6. Press the ÷ key. Notice that you've moved back a few lines.
7. Keep pressing the ÷ key until *unit moving* is at the top of the display.
8. Press the space bar twice to fill the entire screen with code.
9. How can you quickly bring line 31 to the top of the screen? There are two ways: *f3∅* or the X key. Use either of these directives to bring line 31 to the top of the screen.
10. Press the + key a few times to move the code forward.
11. Press the – key a few times to move the code backward.
12. Press Δ (shifted –) to bring *unit moving* to the top of the display.
13. Press Σ (shifted +) to bring the last line of code to the top of the display (line 54).
14. Press the ÷ key three times.
15. Press the space bar once. Press it again; no more lines appear. You are at the end of the code.
16. Experiment with X, ÷, +, –, Σ, and Δ until you can use these directives efficiently.
17. Press BACK to get to the block display.
18. Press BACK again to get to the author mode display.

Section 2 (*r, s, is, out*)

1. On the author mode display, type in your own lesson file name and press NEXT to get to the block display.
2. Create a new block (press *A*, press NEXT at the create-a-block display, and name the block *test*).

3. Enter insert mode. Type:

```
unit     letsmakeamess
at       ØØ5Ø
write mess
```

   This is the code you will edit in this section.

4. Press BACK.
5. Condense the lesson to receive the condense errors (SHIFT-STOP).
6. Press SHIFT-NEXT to edit the first error.
7. Replace line 1 by entering *r1* and pressing NEXT.
8. Press the COPY key once to copy the command. (The command is correct.)
9. Press TAB.
10. Type *mess*. This eliminates the unit tag error by shortening the tag.
11. Press NEXT to bring up the next line. Notice that you are still in replace mode (as noted at the top of the screen).
12. Press the COPY key once to copy the command.
13. Press TAB.
14. Enter Ø15Ø (line 1, character position 5Ø). This is an acceptable coarse-grid position.
15. Press NEXT to edit the next line of code.
16. Press the COPY key once to copy the command.
17. Press TAB.
18. Type *mess*. This puts the appropriate number of spaces in the command field.
19. Press BACK to end the replace mode.
20. Press SHIFT-STOP to condense the lesson. Notice the word that appears in the upper right-hand corner of the screen in position Ø15Ø. This is how the student would see the lesson.
21. Press SHIFT-STOP to go back to the author mode display.
22. Press NEXT to go to the block display.
23. Press *b* to go to block *test*.
24. Delete line 1 of your code (*d1*, SHIFT-HELP).
25. Type *out*. Notice that as soon as you type the *t*, you are taken to the block display.
26. Type *b* to reenter the block. Notice that the line you deleted previously, *unit mess*, is back again; *out* can bring back an accidentally deleted line.

However, if you should insert a number of lines, delete some lines, and then type *out*, the inserted lines will also be eliminated. You will now see this happen using the save option.

27. Type *s3* and press NEXT to save the first three lines.
28. Enter *is3* to insert the saved lines after the third line. Press NEXT.
29. Press the Δ key (shifted –) and then the space bar to make certain you see the entire code. Notice that you now have two *unit mess* instructions.
30. Now type *out*, which takes you to the block display.
31. Type *b* to go into block *test* again. Notice that the three lines you saved and inserted are gone. This shows that if you insert or replace lines of code with the *is*, *i*, or *r* directives and then type *out*, you will not retain the corrected copy. For this reason, you will want to frequently *update* your blocks by returning to the block display page. By updating, you replace old, stored code with new code. The next steps in this section show you how the updating process works.
32. Type *s3* to save the first three lines of code. Press NEXT.
33. Type *is3* and press NEXT to insert the saved code after line 3.
34. Press Δ (shifted –) to look at all six lines of code.
35. Press the BACK key to go to the block display. Your code has now been *updated* (returning to the block display updates your code).
36. Enter block *test* again.
37. Type *out*. (You will return to the block display.)
38. Enter block *test* again. Both units remain because you updated your code before typing *out*.
39. Condense the code by pressing SHIFT-STOP. Notice the condense error: *unit mess* was used twice. This violates the rule for naming units.
40. Press SHIFT-NEXT to edit the error. The second *unit mess* appears at the top of the screen.
41. Type *r1* and press NEXT to replace the -unit- instruction.
42. Press the COPY key *twice* to copy two words.
43. Rename the unit *messier* by adding *ier* to the name *mess*.
44. Press the BACK key.
45. Press Δ (shifted –) to look at the revised code.
46. Press SHIFT-STOP to condense the code. The word *mess* appears in the upper right-hand corner of the screen. When you press NEXT, the second unit instruction will cause the screen to erase and write *mess* again.
47. SHIFT-STOP out of the lesson.

48. At the author mode display, press NEXT. (The PLATO system remembers the lesson you were in.)
49. Type *b* to reenter block *test*.
50. Enter *d6* and press SHIFT-HELP to delete the six lines of code. This will empty your file space for the next activity.
51. Press the BACK key to return to the block display.
52. Press BACK again to return to the author mode display. If you wish to begin the next activity now, you may. If you wish to leave the terminal, press SHIFT-STOP until the *Press NEXT to begin* message appears.

3-G. ENTERING YOUR OWN CODE



Enter and edit code using the PLATO editor.

You should now be able to enter and edit your own code. This activity allows you to do so, using the code concerning primary colors you wrote in unit 2. Sign on to a PLATO terminal as an author and complete these steps:

1. When you see the block display, create a new block in which to enter your code.
2. Enter and edit your code. The editing directives you have learned thus far are:

i	to insert lines of code
d	to delete lines of code
f	to move forward
b	to move backward
se	if you have condensed and received a condense error message, to go directly from the code to the error without condensing the lesson again
X	to move forward one frame
÷	to move back a few lines
+	to move forward one line
-	to move back one line
Σ	to move to the end of the code
Δ	to move to the beginning of the code
COPY	to copy one word when in replace mode
COPY1	to copy one line when in replace mode
r	to replace lines of code
s	to save lines of code
is	to insert saved lines of code
out	to restore a block of code

3. Condense your lesson; if condense errors are shown, edit these errors.
4. Go through the lesson as a student several times. Use the coding directions (this material is located in activity 2-G) to check every possible student response. Be sure to try misspelling words and capitalizing words. Test unanticipated responses. You should receive feedback for these in the first question, but only a *no* judgment in the second question. If you encounter errors, edit the errors, condense the lesson, and check out the lesson as a student once more.
5. When you are satisfied with your program, back out of the lesson.



3-H. LESSON SPACE STRUCTURE AND EXTENDED CORE STORAGE

Identify ways lesson structure affects programming.

Identify ways in which ECS affects programming.

Thus far, when you have entered your file space, you have encountered a block display. You have learned to create a block to program a simple lesson. Eventually, however, you will require more than one block for the lessons you program. Thus, it is necessary to understand the total structure of the lesson space, which consists of:

- Parts
- Blocks
- Words

It is also important to understand how the space available in *extended core storage* (ECS) can influence your programming.

THE STRUCTURE OF A LESSON SPACE

A lesson space contains up to ten *parts.* Each part can contain up to seven *blocks.* The file (lesson) space given to you by the learning center technician probably consists of only *one* part; if you should need more space, you will have to ask the learning center technician for additional parts. Thus far, you have used only two of the blocks in your part: *a. block directory*, which is automatically included in the first block in the first part of a lesson, and *b. test*, in which you have entered code. Thus, you can create up to five more blocks in your lesson part before requesting additional parts.

The number of blocks and, consequently parts, you need is determined by the number of *words* in your lesson. You can enter up to 32Ø words per block. A word consists of ten characters, including spaces between words and excluding blank character positions at the end of lines. Capital letters count as two characters. Do not confuse a programming word with the word on a printed page. The characters *at the* are only part of a programming word; they require six character positions. *Mary Ann* is exactly one word; the M and A count as two characters each for a total of four; the rest of the letters total five character positions, and the space between *Mary* and *Ann* counts as one character position. If a line ends with part of the word, this part is counted as a complete word.

When you have used the 32Ø words in your block, no more characters can be entered; you must create a new block to continue. Whenever possible, start a new block with a new unit; otherwise, people who must enter your lesson to inspect your code will have difficulty interpreting the sequence of your code. (Unit 4 explains how to sequence units to make the best use of your block space.) If it is not possible to begin a new block with a new unit, it is advisable to place a comment in your code explaining that your lesson will cross block boundaries; an appropriate comment would be *$$cross block* if the comment is included at the end of a line, or * *crossblock* if the comment begins a line.

EXTENDED CORE STORAGE

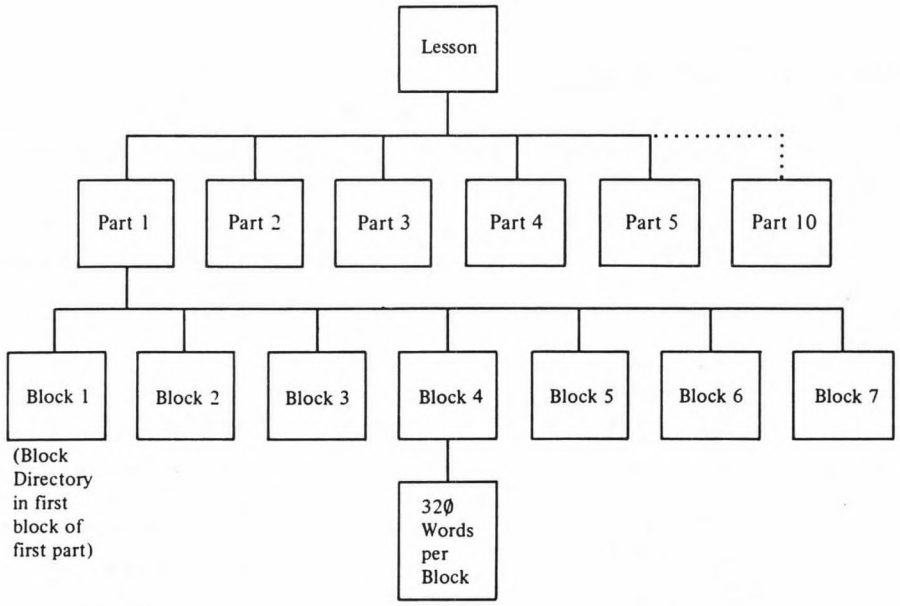
Two major types of storage are used with the PLATO system: disk storage and extended core storage (ECS). Disk storage is cheaper and can hold more than ECS, but requires more time to access information. Therefore, disk is used for long-term storage, while ECS is used for short-term storage. Specifically, ECS is used to store lessons that are being executed, or information that is likely to be modified.

Your lesson code is always stored on disk; while you are editing your code, a copy of the specific block being edited is brought into ECS. The original block is left on disk. When you indicate that you have finished editing the block (by pressing BACK, for example), the updated copy of the block replaces the original on the disk. You have already learned that it is good practice to update your code frequently. Frequent updating minimizes the probability that you will "lose" a great deal of code because of some external system problem, such as a power failure. An exception to the return of the updated block to disk storage results when the *out* directive is used. The *out* directive aborts the return of the updated copy, thus preserving the old copy.

When a lesson is executed, a copy of the original (source) code is condensed. The condensed form is shorter and easier for the computer to interpret. The condensed code is put in ECS, and remains there as long as at least one person is executing the lesson. The source code for the lesson, however, still exists on disk. Thus, an author can be editing a lesson while it is being executed. Because the revised form has not yet been condensed, however, the changes will not occur until after the last user has left the lesson and the lesson is recondensed. Disk storage can hold more than ECS because there is usually considerably more disk available than ECS. However, ECS is required for any lesson being executed. Thus, there are constraints on the amount of ECS that can be used by the condensed code of a lesson. Some constraints are absolute, and cannot be exceeded. Thus, there are guidelines for determining how much ECS is a "fair share" for a lesson. Lessons that require considerably more than their fair share are often called "ECS pigs." You should find the limits and fair share guidelines in use on your system in order to help make the best use of the available system resources.

SUMMARY

The following chart illustrates the structure of a lesson space.



A lesson is stored in ECS while being edited or executed. Because ECS is limited, absolute limits and fair share guidelines exist. Discover these limits and guidelines to help make the best use of your available system resources.

3-I. TEXT EDITING, LESSON THREE



Identify the effects of the *u*, *c*, and *x* search directives.

Editing directives that move the code forward and back work well when you know the exact placement of the material you wish to edit. However, as your code becomes longer and more complex, you may require *search* directives, which can search the code to locate specific material more quickly.

CAI INTRODUCTION

The CAI lesson discusses the use of three search directives:

- The *u* directive, to search for -unit- instructions
- The *c* directive, to search for specified commands
- The *x* directive, to search for a specified character string

Now sign on to a Control Data PLATO terminal using your *student* sign-on. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "Editing Time-Savers 2: Using the Search Directives," press LAB to take the lesson. When you have completed this lesson, read the following summary.

SUMMARY

The *unit* search directive (*u* followed by the name of the unit for which you are searching) is used to find a particular unit. A *u* directive with no name matches any unit; it can be used to advance to the following unit. If a unit directive with a name is entered, the PLATO system will find the correct unit, and the line containing the matched -unit- instruction will appear at the top of the screen. If no match is found, the words *not found* will flash at the top of the screen. If you type the full name of the unit for which you are searching, you will be sure to find the specific unit you want.

The *c* directive (*c* followed by a command) searches for a specified command. For example, *carrow* will locate the next *arrow* command in the block.

The *x* directive (*x* followed by a string of characters) searches for any occurrence of a specified character string. For example, *xthe* will locate the next occurrence of the letters *the*.

If you must use the same directive many times, the PLATO system will remember the last directive used and repeat it if you press NEXT without typing anything.

All search directives can search an entire lesson, which is helpful if you can't remember which block contains the information you need. To initiate a lesson search, type the directive you would use to search a block, but press SHIFT-NEXT instead of NEXT. Lesson searches should not be overused; they can, however, save a great deal of time when properly used.

CHECK YOUR UNDERSTANDING

Now use the *u*, *c*, and *x* directives to search the code in your file space. Practice until you are comfortable with these directives.

3-J. EDITING HELP



Identify procedures for receiving help on editing.

If you ever need help while editing, a help sequence is always available on the system to provide information. You can access an editing help sequence by pressing HELP on the block display or on any editing display.

When you press HELP, you will see this display:

```
Editing HELP

1.  Introduction to Editing

2.  Block Directory Options

3.  Line Display Options
```

Now sign on to a PLATO terminal as an author. Access the help sequence from the block display. Read the information presented in this sequence; it provides a good review for the information presented thus far in this course. Some of the information will be new to you; you may skim over the new material if you wish.

4

Lesson Sequencing and Display Control

This unit discusses instructions used to sequence the units of a lesson. Lesson sequencing instructions make use of the PLATO system's branching capabilities. This unit also presents instructions used to manipulate displays within a unit. Practice in the use of each instruction is provided.

INSTRUCTIONAL OBJECTIVES

After completing this unit, you should be able to:

- Identify the effects of the -next-, -next1-, -back-, -back1-, and -stop- instructions
- Identify the effects of the help sequence instructions: -help-, -help1-, -data-, -data1-, -lab-, and -lab1-
- Given descriptions of help sequences, identify main units and base units
- Identify the effects of the -do-, -join-, and -jump- instructions
- Given descriptions of sequences of units, identify auxiliary units
- Identify the effects of the -helpop-, -dataop-, and -labop- instructions
- Identify the effects of -pause-, -erase- with a tag, -inhibit- with an *erase* tag, -erase- with no tag, and -end- with a *lesson* tag
- Given programmable-ready material, write a program using the instructions presented in this unit

LEARNING ACTIVITIES

_____ 4-A. Lesson Control With Function Keys. Text/Exercise/CAI: This activity explains five instructions used to sequence units: -next-, -next1-, -back-, -back1-, and -stop-.

_____ 4-B. Help Sequences. Text/Exercise/CAI: This activity discusses instructions frequently used for help sequences, including -help-, -help1-, -data-, -data1-, -lab-, and -lab1-. It also defines the terms *main unit*, *base unit*, and *base marker*.

_____ 4-C. Author-initiated Branching. Text/Exercise/CAI: This activity discusses three instructions used for author-initiated branching (branching over which the student has no control). The -do-, -join-, and -jump- instructions are explained, and the term *auxiliary unit* is defined.

_____ 4-D. The Helpop Family. Text/CAI: This activity presents the -helpop-, -dataop-, and -labop- instructions. These instructions are used to supply help in a current unit without erasing the screen.

_____ 4-E. Display Control Instructions. Text/Exercise/CAI: This activity explains four instructions used to manipulate displays: -pause-, -erase- with a tag, -inhibit- with an *erase* tag, and -erase- with a blank tag. It also discusses the use of the -end- instruction to end a lesson.

_____ 4-F. Comprehensive Practice. Text/CAI: This activity provides programmable-ready material that calls for the use of the instructions in this unit.

TESTING DIRECTIONS

Now sign on to a Control Data PLATO terminal. If you are very familiar with the subject of lesson sequencing and display control you may wish to test out of some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.

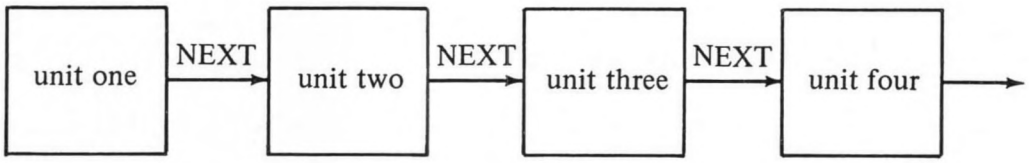
4-A. LESSON CONTROL WITH FUNCTION KEYS



Identify the effects of the -next-, -next1-, -back-, -back1-, and -stop- instructions.

Units are the building blocks of PLATO lessons. A unit usually performs a complete instructional step, such as putting a display on the screen. Each unit consists of a group of PLATO Author Language instructions that begins with the -unit- instruction and ends when the next -unit- instruction signifies the start of another unit.

The simplest structure of a lesson containing more than one unit is a consecutive sequence of units:



This structure requires no sequencing efforts on the part of the author because the PLATO system assumes, unless otherwise specified, that the next unit in sequence will be executed when the NEXT key is pressed.

Often, however, you will want to sequence units in other ways. For instance, you may wish to organize units in a way that best uses the space in your lesson blocks. This activity discusses instructions that can be used to sequence units: -next-, -next1-, -back-, -back1-, and -stop-. It includes a CAI lesson that simulates the effects of these instructions, and it discusses important differences between the -next- instruction and the other four sequencing instructions.

SEQUENCING INSTRUCTIONS

If you want units to be executed in a sequence other than consecutive, any one of five instructions can be used:

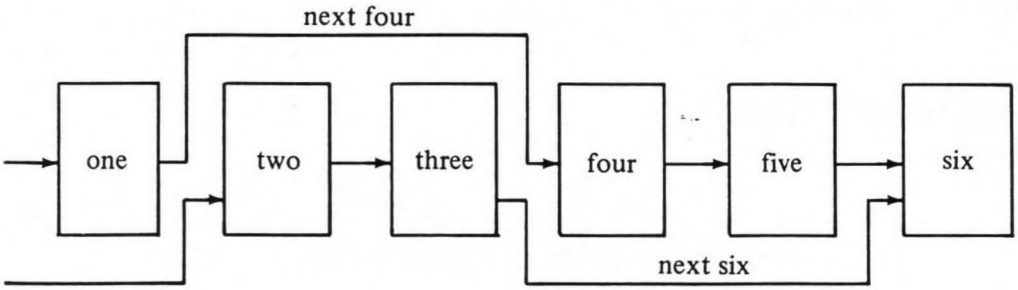
-next- (unit executed when the student presses NEXT)
-next1- (unit executed when the student presses SHIFT-NEXT)
-back- (unit executed when the student presses BACK)
-back1- (unit executed when the student presses SHIFT-BACK)
-stop- (unit executed when the student presses STOP)

These instructions all use the same format:

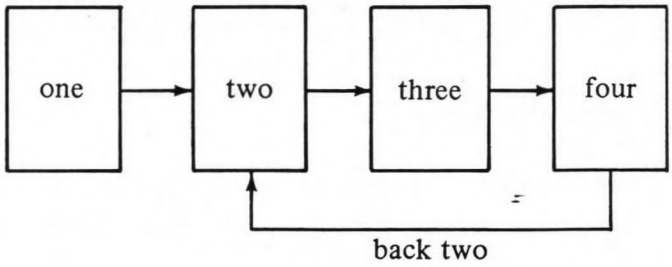
```
next     NAME OF UNIT
next1    NAME OF UNIT
back     NAME OF UNIT
back1    NAME OF UNIT
stop     NAME OF UNIT
```

name of unit to be executed if function key specified in the command is pressed

They can all be used to branch the student over units. For example:



These instructions can also be used to review units previously completed:



Combinations of these instructions can be used to provide a number of different sequencing options in a single unit; this is the reason so many sequencing instructions exist. When function keys are activated using any of these five instructions, it is often advisable to insert a -write- instruction telling the students what to do. For example:

```
unit     five
back     three
at       810
write    Who wrote TWICE-TOLD TALES?
at       3010
write    Press BACK to review.
```

Because the student must participate in the branching effort, it is advisable to sequence units using keys that give clues to what will occur. For example:

- BACK is often used to review the previous unit.
- SHIFT-BACK (called for by the -back1- instruction) is often used to review a sequence of units.
- NEXT is often used to move to the next logical unit.
- SHIFT-NEXT (called for by the -next1- instruction) is often used to skip over a number of units.
- STOP is often used to terminate a section of a lesson, or the entire lesson.

CAI INTRODUCTION

This CAI lesson simulates the execution of the -next-, -next1-, -back-, and -back1- instructions. The -stop- instruction is *not* simulated; it is actually used within the lesson to allow you to terminate the exercise sections.

Now sign on to a PLATO terminal using your *student* sign-on. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "The Function Keys: next, back, next1, back1," press LAB to take the lesson.

THE UNIQUE -next- INSTRUCTION

Though all five instructions are formatted the same way and perform essentially the same function, the -next- instruction differs from the others in two important ways:

1. If the NEXT key is pressed at the completion of a unit, the student will "fall through" to the next consecutive unit, even though no -next- instruction has been coded (the NEXT key is automatically active). If a -next- instruction *has* been coded, the student will be branched to the unit specified in the tag upon completion of the current unit. With the -back-, -back1-, -stop-, or -next1- instructions, however, an instruction must be *coded* to activate the BACK, SHIFT-BACK, STOP or SHIFT-NEXT keys. Otherwise, nothing will happen when the students press these keys.

2. Branching via the -next- instruction occurs only upon *completion* of the current unit, regardless of the placement of the instruction. With the other four instructions, branching can occur at *any* time during the unit, depending on the placement of the instruction. For example, if you code a unit to include a question, you may wish to allow the student to press BACK to review upon entry of an *in*correct answer, but not *before* answering or upon entry of a *correct* answer. In this case, your code would look like this:

```
unit     test
at       Ø51Ø
write    What is the capital of Wisconsin?
arrow    Ø81Ø
answer   Madison
no
back     Wis
at       321Ø
write    Press BACK to review
at       111Ø
write    The name of the city begins with M.
endarrow
```

If this code is used, the *Press BACK to review* message will remain even if the student presses NEXT to reenter an answer; when an instruction resulting in a *no* judgment is encountered, only the last -write- display is erased when the student presses NEXT to try again. In most cases, however, you will want the BACK, SHIFT-BACK, STOP, and SHIFT-NEXT keys active throughout the unit; at these times, place the instruction under the -unit- instruction to make code inspection easier. With all instructions, you will have to include a unit in your code to which the instruction can branch; otherwise, a condense error will result.

CHECK YOUR UNDERSTANDING

Section 1

Directions: Fill in the blanks with the correct answers.

1. If a student presses the BACK key in a unit containing a *back revu* instruction, to what unit will the student branch?

   _____

2. If a student presses the NEXT key in a unit containing a *next info* instruction, to what unit will the student branch?

   _____

3. In what two ways does the -next- instruction differ from the other four sequencing instructions?

   a. _____

   b. _____

4. What is the potential problem in this code?

```
unit    test
back    review
at      Ø51Ø
write   Who invented movable type?
arrow   71Ø
answer  Johann Gutenberg
no
write   He was a German inventor.
endarrow
```

Answers

1. unit revu
2. unit info

3a. If no -next- instruction is coded, the student will fall through to the next unit.

3b. The -next- instruction can be placed anywhere in the code with the same results; the student will branch upon completion of the unit.

4. No -write- instruction has been included to provide instructions for pressing BACK to review. Such documentation should be present somewhere in the lesson. If BACK is consistently used for review throughout the lesson, documentation to this effect may be included at the beginning of the activity, thus avoiding the need to write it on each display. If the availability of BACK is sporadic, each use should be documented at the time of availability.

Section 2

Directions: Write the letter of the *best* answer in each blank.

_____ 1. When will a NEXT key press branch the student to the unit named in the tag of the -next- instruction?

   a. During the entire unit
   b. Before arrow processing only
   c. After arrow processing only
   d. Upon completion of the unit

_____ 2. When will a STOP key press branch the student to the unit named in the tag of this -stop- instruction?

```
unit     four
stop     out
at       1020
write    How many inches make a foot?
arrow    1220
answer   (12,twelve)
no
write    Try again.
endarrow
```

a. During the entire unit
b. Before arrow processing only
c. After arrow processing only
d. Upon completion of the unit

_____ 3. When will a BACK key press branch the student to the unit named in the tag of the -back- instruction in this code?

```
unit     cow
at       0510
write    What animal gives us milk to drink?
arrow    0710
answer   cow
no
back     milk
at       3210
write    Press BACK to review.
at       0910
write    This animal says, "Moo."
endarrow
```

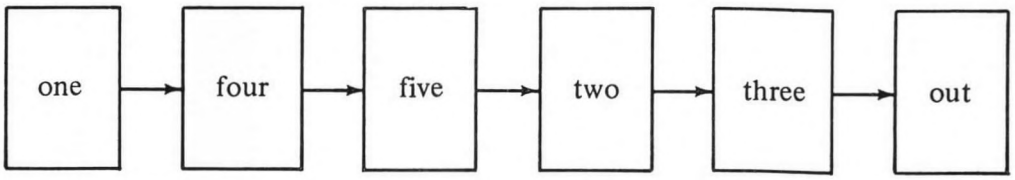
a. During the entire unit
b. Before arrow processing only
c. After arrow processing only
d. After the student has entered a response other than *cow*

Answers

1. d 2. a 3. d; an unanticipated response must be entered if the -back- instruction is to be executed. If the correct answer is given, the PLATO system proceeds to the -endarrow-.

Section 3

Directions: The diagram shows the order in which the units have been physically sequenced. Use this diagram to answer the following questions.



1. What instruction should be placed in *unit one* if the next logical unit in the sequence is *unit two*?

   _____

2. Is a -next- instruction necessary to allow the student to see *unit three* after *unit two*?

   _____

3. What instruction is the most logical to use in *unit five* if you want the student to be able to review *unit four* while in *unit five*?

   _____

4. What instruction would be most logical to use in *unit out* (the last unit) to allow the student to begin the lesson again?

   _____

Answers

1. next two
2. no
3. back four
4. back1 one

These answers are based on the *logical use* concept. There may be times when these instructions must be used in other ways.

Section 4

Directions: You should now be able to use the -next-, -next1-, -back-, back1-, and -stop- instructions to sequence units within your lessons. If you wish to practice using these instructions:

1. Write units of code on a subject familiar to you. Include the lesson sequence instructions you learned in this activity.
2. Enter your code in a new block in your file space.
3. Condense the code. If condense errors are shown, eliminate these errors. If you wish to *prevent* condensation of other blocks, follow this procedure:
   a. At the block display, press the SHIFT key and the letter *p*.
   b. Type the letter of the block you do not wish to condense. A dash will appear in front of that block. Press NEXT.
   c. If, at a later time, you wish to condense this block, use the same process to remove the dash (SHIFT-p and the letter of the block).
4. Go through the lesson as a student to see if it executes as you anticipated.
5. Eliminate any errors.
6. Delete the block when you finish, if you have no other empty blocks in your file space.

4-B. HELP SEQUENCES



Identify the effects of the help sequence instructions: -help-, -help1-, -data-, -data1-, -lab-, and -lab1-.

Given descriptions of help sequences, identify main units and base units.

Activity 4-A discusses unit sequencing instructions, which can be used to allow students to move forward and backward in a series of units. Unit sequencing instructions alone, however, do not make full use of the PLATO system's branching capabilities. More extensive branching can be accomplished using *help sequences*.

This activity defines the purposes of help sequences and discusses instructions (and their related function keys) frequently used for help sequences. It also includes a CAI lesson that simulates the help sequence flow, discusses lesson sequencing instructions used within help sequences, and explains how to override the automatic sequencing using the -base- instruction.

THE PURPOSE OF HELP SEQUENCES

The term *help sequence* results from the fact that the keys associated with the help sequence instructions are often used to allow the student to receive help in the form of additional information, review information, or additional practice. Help sequences, however, can perform other functions, as you will see later in this activity, when a more complete definition can be given. You saw an example of a help sequence in activity 3-J, when you pressed HELP on the block display to see information on editing.

HELP SEQUENCE INSTRUCTIONS

Instructions used for help sequencing include -help-, -help1-, -lab-, -lab1-, -data-, and -data1-. All have the same general format:

```
help     NAME OF UNIT
help1    NAME OF UNIT
lab      NAME OF UNIT
lab1     NAME OF UNIT
data     NAME OF UNIT
data1    NAME OF UNIT
         ↑
         └─ name of first help unit to be executed
```

The instructions will be executed only when the student presses the appropriate function keys (HELP for the -help- instruction, SHIFT-HELP for the -help1- instruction). Thus, as with unit sequencing instructions, it is important to include a -write- instruction signaling that a help sequence is available.

Although help sequencing instructions can be used interchangeably, it is important to remember that students often associate the names of function keys with their effects. For example:

- HELP often leads to units of additional or review information needed by some, but not all, students.
- LAB often leads to exercises. (It is an abbreviation of *laboratory*.)
- DATA often leads to information relative to a specific situation, such as a chart or data needed to answer a question.

Thus, you may wish to use a combination of help sequences in a given unit. For example, students in a unit might be allowed to press LAB to get an exercise *and* HELP to receive additional information before completing the exercise.

If help is available throughout a unit, it is best to place help sequence instructions near the beginning of the unit code to make code inspection easier. Otherwise, place the help sequence instruction at the point in the code where you want it to execute.

This code demonstrates the use of the -help- instruction:

```
unit    recipe
help    seconds
at      1515
write   HOW TO MAKE CREPES
at      1818
write   The recipe for crepes is simple.  Just combine:
        1 cup flour
        1 cup milk
        2 eggs
        1 tbsp. brandy
        Heat an iron skillet until it is very hot. Pour
        in just enough batter to cover the bottom of
        the pan. Cook until the top side is firm.
at      3Ø32
write   Press HELP for a smaller recipe.
at      3232
write   Press NEXT to continue.
***
more units of code
***
```

```
unit    seconds
at      1Ø15
write   Need a smaller recipe? Here it is:
.
.
.
```

If students press the HELP key in *unit recipe*, they will be branched to *unit seconds*. Upon completion of the help sequence, they will branch back to the beginning of *unit recipe*.

If students do *not* press HELP in *unit recipe*, they will continue to the next unit in the physical sequence upon completion of the current unit. They will not see *unit seconds*.

CAI INTRODUCTION

This CAI lesson simulates the execution of units in a help sequence. Before beginning this lesson, it is important to have a rudimentary understanding of the concepts of *main unit*, *base unit*, and *base marker* in order to understand the execution process.

All units you have coded thus far in this course are *main units*; you have accessed main units by pressing function keys. A main unit, such as those you have seen, becomes a *base unit* when a function key associated with a help sequence instruction has been pressed. A help sequence is brought about in this way:

1. The student is in a main unit.
2. The student presses a function key associated with a help sequence instruction.
3. The main unit becomes a base unit.
4. The student enters the help unit specified by the help sequence instruction (a new main unit).

It is important to understand the main unit and base unit concepts because of a third concept, the *base marker*. When a student begins a help sequence, the PLATO system marks the base unit from which the sequence was accessed with a base marker. This marker ensures that the student will be returned to the appropriate base unit upon completion of the help sequence or depression of the BACK or SHIFT-BACK keys. (These keys are automatically activated when a student is in a help sequence unless instructions specifying otherwise are included in a unit of the help sequence.) The base marker automatically specifies the unit to which the student will return from a help sequence.

The lesson sequencing instructions presented in activity 4-A (-next-, -next1-, -back-, -back1-, -stop-) do *not* set the base marker; for example, if the -back1- instruction is used to move the student from *unit five* to *unit two*, the PLATO

system will execute units two through five again. The student will *not* be returned to *unit five* if the BACK key is pressed. Thus, the true definition of a help sequence instruction is *an instruction that causes a base marker to be set.*

Now sign on to a PLATO terminal using your *student* sign-on. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "HELP Sequences," press LAB to take the lesson. When you finish the CAI lesson, continue with the next section of this activity in your text before you attempt to incorporate a help sequence in your own code.

LESSON SEQUENCE INSTRUCTIONS WITHIN A HELP SEQUENCE

The -end- instruction is the most important lesson sequence instruction within a help sequence. This instruction *signals the end of the help sequence and terminates its execution.* When the NEXT key is pressed, the student returns to the base unit; if no -end- is included, the PLATO system continues to execute units as directed by the unit sequencing instructions, while remaining in a help sequence; the base marker is still set.

The format of the -end- instruction in a help sequence is:

```
end        BLANK TAG
```

If the help sequence unit is used for no purpose other than help, the -end- instruction is placed at the end of the code of the last help unit. Code written after the -end- instruction will not be seen by the student because -end- terminates execution of the instructions in the help sequence.

Earlier in this activity, you learned how a help sequence is brought about. The following steps detail how the help sequence is concluded:

1. The student is in a main unit.
2. The student presses a function key associated with a help sequence instruction.
3. A base marker is set. It names the main unit as a base unit.
4. The student enters a new main unit (named by the help sequence instruction).
5. The student continues in the help sequence until an -end- instruction is encountered, or until either the BACK or SHIFT-BACK keys are depressed.
6. If the -end- instruction is encountered, execution terminates until the NEXT key is pressed.
7. When the NEXT key is pressed, the student returns to the *beginning* of the base unit.
8. The base marker is cleared; the student is no longer in a help sequence.

When you include more than one unit in a help sequence, you may wish to include the -next-, -next1-, -back-, -back1-, or -stop- instructions in your help sequence; these instructions allow more student control within the help sequence. However, the use of these instructions to provide student-controlled branching does not alter the fact that the base marker has been set; the student is still in the help sequence.

If you include the -back- or -back1- instruction as a lesson sequencing instruction within a unit in a help sequence, you override the automatic return to the base unit if the student presses those keys while in that unit of the help sequence. Figure 9 illustrates the effects of lesson sequencing instructions within a help sequence. The *back help* instruction in *unit help1* allows the student to press BACK to see *unit help* again. The *back help1* instruction in *unit help2* allows the student to review *unit help1*. Thus, lesson sequencing instructions produce the same results in a help sequence as they do under other circumstances.

Now study figure 9 to see how lesson sequencing instructions affect the automatic sequencing. A press of the SHIFT-BACK key is now required to return the student to the base unit from *unit help1*; SHIFT-BACK will also return the student to the base unit from *unit help* or *unit help2*. The automatic BACK in all help units except the first has been overridden by the -back- instructions.

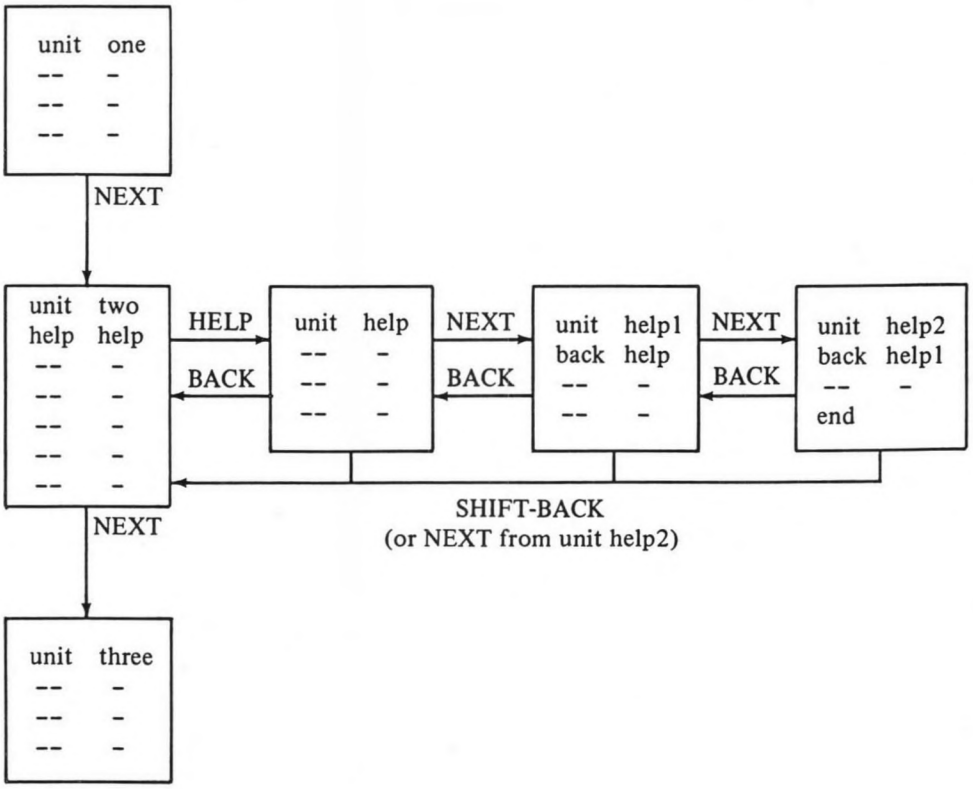


Figure 9. Lesson sequencing in a help sequence

Notice the -next- instruction in *unit help2* of figure 10. Even though that instruction directs the logical flow to *unit three* upon a NEXT key press, it will be ignored if the base marker has been set, indicating that the student is in a help sequence. The -end- instruction in *unit help2* will override the -next- instruction *because a base marker has been set*. The student will be returned to *unit two*, the base unit, if the NEXT key is pressed at the termination of the help sequence.

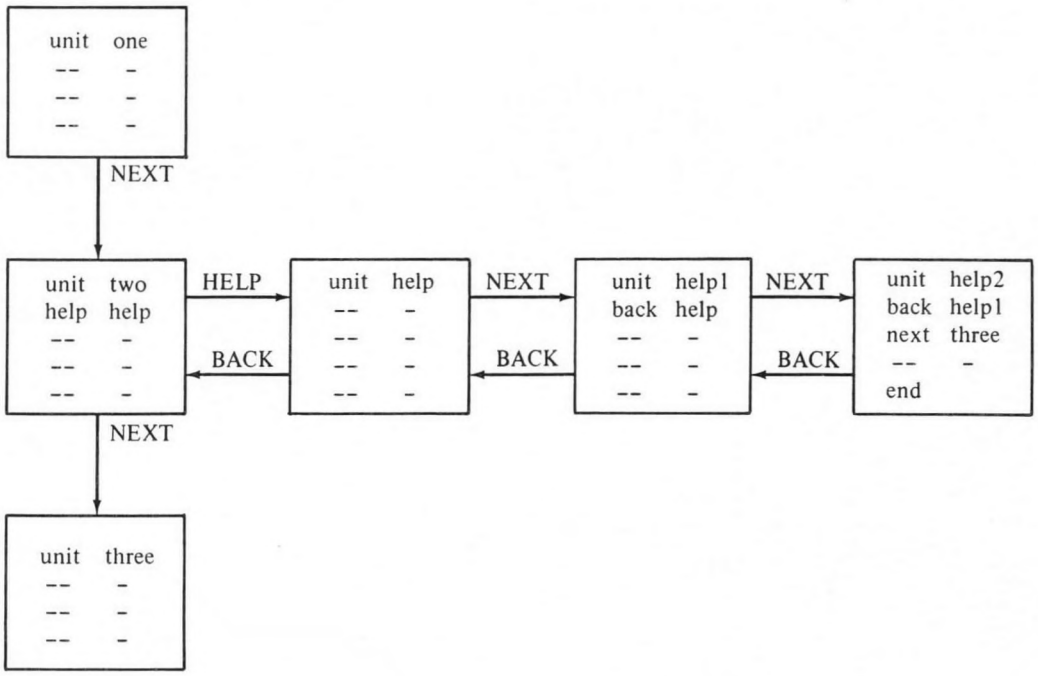


Figure 10. A useless -next- instruction

When you are using existing units for dual purposes (for example, a unit used both in the main sequence and in the help sequence), it is important to watch lesson sequencing instructions carefully. Be sure the instructions branch students to acceptable units.

OVERRIDING AUTOMATIC HELP SEQUENCING

Help sequences can save a great deal of programming time and code, and will automatically return the student to the base unit. There may be times, however, when you want to "turn off" the help sequence after the student has entered it; in other words, you may want to clear the base marker. The -base- instruction, with a blank tag, will clear this marker so the student will not return to the base unit. The regular sequencing instructions are in effect; automatic sequencing no longer operates because the student is no longer in a help sequence.

The -base- instruction should be included at the point at which you want to terminate the help sequence. For example, in figure 11, if -base- is included in *unit help*, nothing will happen if BACK is pressed. If the -base- instruction is included in *unit help2*, BACK will take the student to *unit help1*, but if NEXT is pressed, the student will be taken to *unit three*. The -base- instruction overrides the end instruction.

If a tag is included in the -base- instruction, the instruction will *reset* the base marker to the unit specified in the tag. Automatic return sequences are still active, because a base marker is set; however, the automatic returns will return to the new base unit.

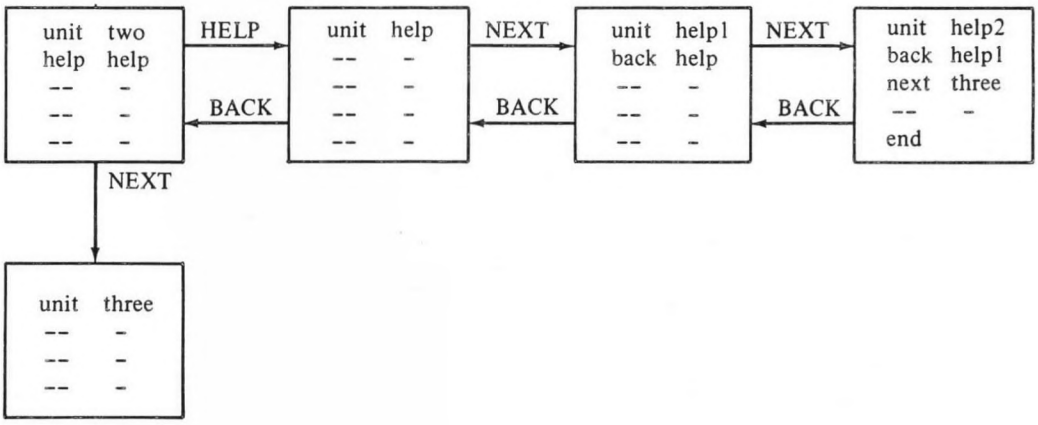


Figure 11. A visualization of the effects of the -base- instruction

CHECK YOUR UNDERSTANDING

Section 1

Directions: Answer the questions by filling in the blanks.

1a. Which are the most logical keys for the student to press to see units of additional or review information needed by some, but not all, students?

_____

1b. What two instructions activate these keys?

_____

2a. What are the most logical keys for the student to press to receive additional exercises?

_____

2b. What two instructions activate these keys?

_____

3a. What are the most logical keys for the student to press to receive a chart needed to answer a question?

_____

3b. What two instructions activate these keys?

_____

4. Can more than one help sequence instruction be included in a unit?

_____

5. Where does the student return upon completion of the help sequence?

_____

6. What is the true definition of a help sequence instruction?

_____

7a. What instruction must be included in a help sequence to automatically return the student to the base unit when the help sequence is completed?

_____

7b. Where is the -end- instruction placed, if the help sequence is used for no other purpose than help?

_____

Answers

1a. HELP, SHIFT-HELP

1b. -help-, -help1-

2a. LAB, SHIFT-LAB

2b. -lab-, -lab1-

3a. DATA, SHIFT-DATA

3b. -data-, -data1-

4. Yes, help sequences may be required for more than one purpose in a single unit.

5. To the base unit (the unit from which the student accessed the help sequence)

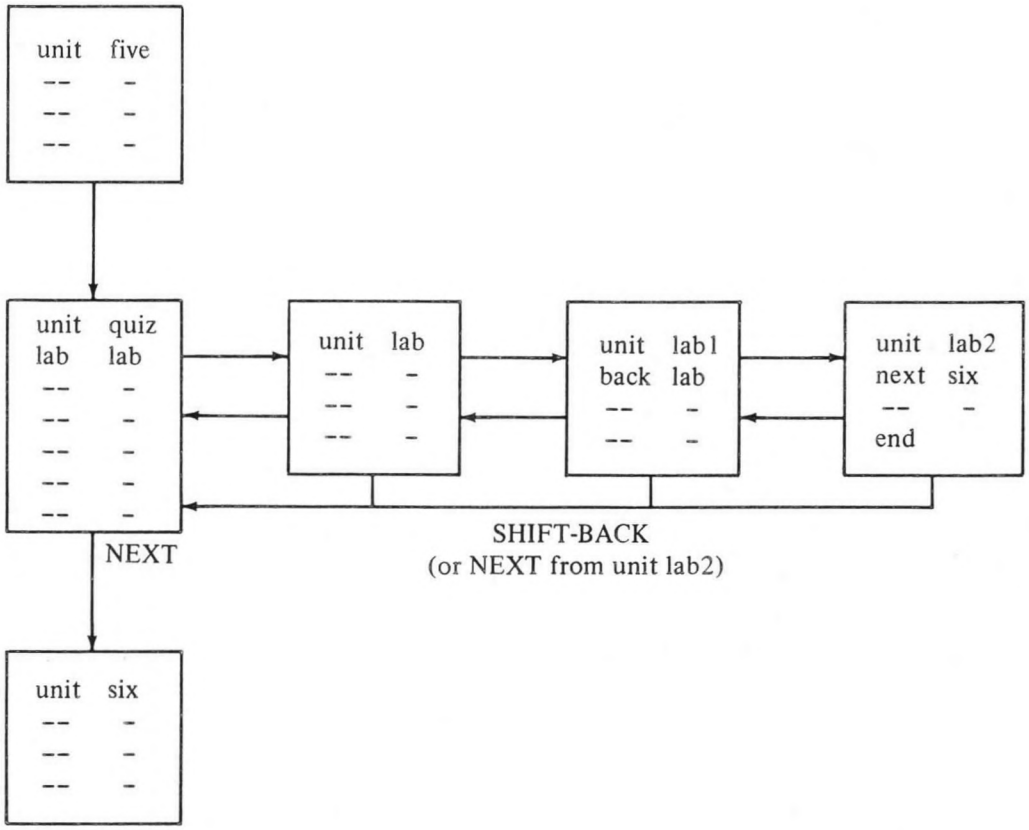
6. An instruction that causes a base marker to be set

7a. -end-

7b. At the end of the last help unit

Section 2

Directions: Use the following diagram to answer the questions.



1. What key must the student press to access the help sequence while in *unit quiz*?

   _____

2. To what unit will the student branch if BACK is pressed while the student is in *unit lab1*?

   _____

3. If *unit lab2* immediately follows *unit lab1* in the code, what key must the student press to move from *unit lab1* to *unit lab2*?

   _____

4. How many function keys are active at the conclusion of *unit lab2* that will take the student to the base unit? What are they?

_____

5. If the -base- instruction with a blank tag is included in *unit lab2*, where will a NEXT key press take the student?

_____

Answers

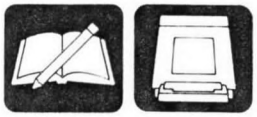
1. LAB
2. unit lab
3. NEXT
4. 3, BACK, SHIFT-BACK, NEXT
5. unit six

Section 3

Directions: You should now be able to use the -help-, -help1-, -lab-, -lab1-, -data-, and -data1- instructions to include help sequences in your lessons. If you wish to practice using these instructions:

1. Write enough units of code, using the help sequence instructions, to enable the units to execute. (You may add new units to previously coded lessons if you wish.)
2. Enter the code in a new block in your file space.
3. Condense the code; eliminate condense errors. (Prevent condensation of other blocks by pressing SHIFT-p and the letter of the appropriate block on the block display.)
4. Go through the lesson as a student to see if it executes as you anticipated.
5. Eliminate any errors.
6. Delete the block when you finish, if you need the space for the next activity.

4-C. AUTHOR-INITIATED BRANCHING



Identify the effects of the -do-, -join-, and -jump- instructions.
Given descriptions of sequences of units, identify auxiliary units.

The instructions you have learned thus far allow students to access main units by choice, through the use of function keys. This is called *student-initiated branching.* The instructions presented in this activity, -do-, -join-, and -jump-, are used for *author-initiated branching*; the student has no choice. In fact, the student may not even realize that branching has occurred.

THE -do- INSTRUCTION

Use the -do- instruction when you wish to insert one unit of code *into* another (when you require a unit within a unit). For example, you may have coded a set of directions in one unit, and may wish to use these directions again in another unit. Recoding the directions is an inefficient use of programming time and storage space. The -do- instruction, using this format

```
do      NAME OF UNIT TO BE INCLUDED
```

will find the unit you require and include it in the unit you are currently coding. The following is a sample of code that includes the -do- instruction:

```
unit    text
at      61Ø
do      direct
at      1Ø1Ø
write   Does the fruit of the peanut plant grow
        above or below ground?
arrow   131Ø
  ⋮

***
more units of code
***
unit    direct
write   Answer the following questions by typing the
        correct answer.
```

In this case, the PLATO system begins to execute *unit text*. When it encounters the -do- instruction, it "attaches" the code of *unit direct* to the current unit. When *unit direct* has been executed, the PLATO system executes the remainder of *unit text*, beginning with the -at- instruction, and then proceeds to the next unit in the sequence.

Because the unit retrieved by the -do- instruction becomes part of the unit currently being coded, it is called an attached, or *auxiliary*, unit. The access of an auxiliary unit does not cause a full screen erase. At this point, the term *main units* can be more precisely defined: A main unit is any unit that is not an auxiliary unit. The main unit concept is important for instructions presented later in this course.

The -do- instruction has an important limitation: Avoid using -do- to retrieve a unit that includes judging commands (-arrow-, -specs-, -answer-, and so forth) unless the unit contains everything from the -arrow- to the -endarrow-. The -do- instruction is executed only in regular state; your judging commands may not execute in the way you anticipate.

THE -join- INSTRUCTION

The -join- instruction is also used to attach auxiliary units to main units. If you wish to include an incomplete arrow processing unit within a unit you are coding, use the -join- instruction. For example, you may wish to ask a series of true-false questions, and wish to "join" units when appropriate:

Main Units

```
unit     suez
at       510
write    The Suez Canal is in Africa.
arrow    813
join     falseok
endarrow
```

Auxiliary Units (Units to be Joined)

```
unit     falseok
answer   false
wrong    true
no
write    Please answer
         true or false.
```

```
unit     ivory
at       510
write    The Ivory Coast is in Africa.
arrow    813
join     trueok
endarrow
```

```
unit     trueok
answer   true
wrong    false
no
write    Please answer
         true or false.
```

In this case, the PLATO system will retrieve the appropriate unit, execute it, and then continue execution of the main unit. The -join- instruction used in this way can save a considerable amount of programming time and storage space.

The -join- instruction is executed in all execution states; occasionally, a -join- instruction will execute in a way other than the way you intended. Thus, whenever you include a -join- instruction in your code, check your program as a student to be sure the results are acceptable.

If the results are unsatisfactory, check for errors in the code of both the main unit and the auxiliary unit. If there is no problem with the code that you can find at this time, revise your code so the auxiliary unit is a complete arrow processing unit; this enables you to use the -do- instruction to attach the unit.

THE -jump- INSTRUCTION

Use the -jump- instruction when you want the student to branch immediately to a new main unit; this will cause a full screen erase, as with all branches to new main units. For example, if you code a unit that asks a question, you may require a student to receive additional units of information if an incorrect response is given; this can be accomplished by jumping to new main units. However, you may want a student who responds correctly to continue with the current unit. If the -jump- instruction is placed correctly in the code, you can achieve these results.

The -jump- instruction uses this format:

```
jump    NAME OF UNIT TO WHICH STUDENT WILL JUMP
```

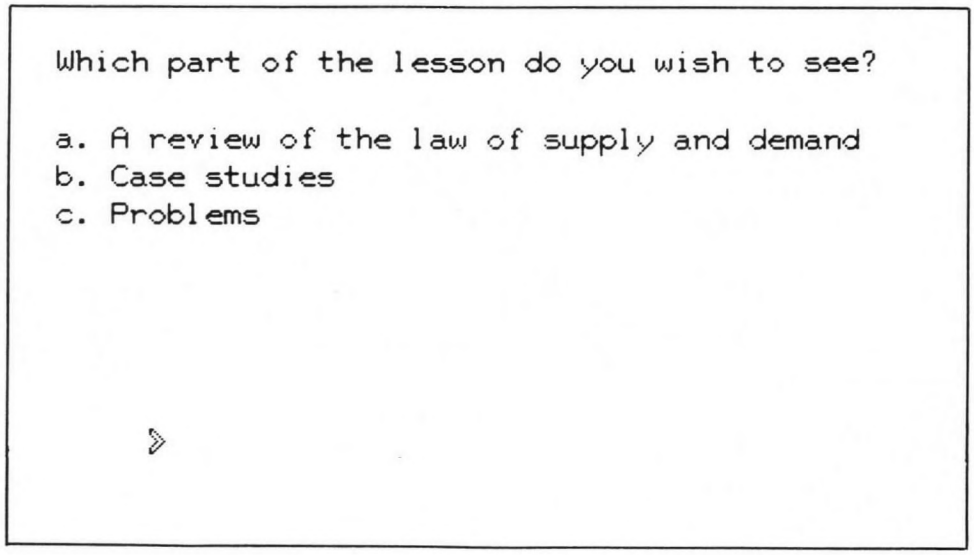
The code for the previous example could look like this:

```
unit    test
at      5Ø8
write   Who was the first president of the United States?
arrow   8Ø8
answer  Washington
at      11Ø8
write   Very good
no
jump    wash
endarrow
***
unit    test2
at      5Ø8
write   ........
  .
  .     MORE INSTRUCTIONS
  .
```

```
***
more units of code
***
unit    wash
at      500
write   George Washington was the first president of the
        United States.  He was also ...
***
```

The students who answer correctly continue with *unit test2* upon completion of *unit test*. The students who answer incorrectly are jumped to *unit wash*; they will continue in the sequence as directed by the new main unit.

A second example of an appropriate use of the -jump- instruction concerns multiple-choice branching; you may wish to allow the student to choose from a list of several different paths. For example:



This can be coded, using the -jump- instruction, to allow the students to branch to the sections of their choice:

```
unit    choice
at      1010
write   Which part of the lesson do you wish to see?
at      1210
write   a. A review of the law of supply and demand
        b. Case studies
        c. Problems
arrow   2015
```

```
answer   a
jump     review
answer   b
jump     case
answer   c
jump     prob
endarrow
```

Students branch to *unit review* when the *a* key is *entered* (the *a* key and the NEXT key are pressed), to *unit case* when the *b* key is entered, or to *unit prob* when the *c* key is entered.

CHECK YOUR UNDERSTANDING

Section 1

Directions: Answer the questions in the blanks provided.

1. Which two instructions can be used to execute a unit within a unit?

   _____

2. What is the term given to units that become part of the unit currently being coded?

   _____

3. What is a main unit?

   _____

4. In what states is the -do- instruction executed?

   _____

5. What instruction should be used to include incomplete arrow processing units as auxiliary units?

   _____

6. What should you do to see if a -join- instruction executes in the way you had anticipated?

   _____

7. Which instruction branches the student to a new main unit?

   _____

Answers

1. -do-, -join-
2. Auxiliary unit
3. Any unit other than an auxiliary unit
4. Only in regular state
5. -join-
6. Take the lesson as a student
7. -jump-

Section 2

Directions: You should now be able to use the -do-, -join-, and -jump- instructions in your lessons. If you wish to practice using these instructions:

1. Write enough units of code, using these instructions, to enable the units to execute; you may expand on the sample code contained in this activity if you wish.
2. Enter the code in a new block in your file space.
3. Condense the code; eliminate condense errors.
4. Go through the lesson as a student to see if it executes as you had anticipated.
5. Eliminate any errors.
6. Delete the block when you finish, if you need the space for the next activity.

4-D. THE HELPOP FAMILY



Identify the effects of the -helpop-, -dataop-, and -labop- instructions.

When the student begins a new main unit, the last main unit is erased. This is true for help units also; when the student begins a help unit, the base unit is erased. If, however, you wish to supply help, data, or exercises on the *same* display as the unit currently being executed, instructions in the helpop family can be used. These instructions are:

-helpop-
-dataop-
-labop-

THE -helpop- FAMILY

All instructions in the -helpop- family have the same general format:

```
helpop   NAME OF UNIT
dataop   NAME OF UNIT
labop    NAME OF UNIT
```

name of unit to be displayed

The *op* at the end of the word stands for *o*n *p*age. For example, -helpop- means *help on page*.

Do not confuse these instructions with help sequence instructions; -helpop- family instructions do *not* set a base marker. They present a new unit as an auxiliary unit without erasing the current unit, after which the student continues with the remainder of the old main unit.

Thus, -helpop- family instructions function much like -do- and -join- instructions, except that the student controls the branching using the associated function keys (HELP to execute -helpop- instructions, DATA to execute -dataop- instructions, and LAB to execute -labop- instructions). The students should be told when these function keys are active.

This sample code uses the -helpop- instruction. The -labop- and -dataop- instructions are used in the same way, except that they are usually used to present information the student will associate with the LAB or DATA keys.

```
unit     dog
at       505
write    Two common breeds of Irish setter are the show
         setter and the _____ setter.
at       3015
write    HELP is available.
arrow    808
helpop   setters
answer   field
wrong    (chestnut,red,mahogany)
no
endarrow
***
more units of code
***
unit     setters
write    The show setter is the long-haired variety of
         setter most often visualized when the term
         Irish setter is heard. The other breed of setter,
         the field setter, is used primarily for hunting;
         its short hair is less likely to become matted
         with burrs.
```

Help is available as soon as the -arrow- instruction is encountered; if the student presses HELP before entering the correct answer, the correct answer must be typed before unit execution can continue. When HELP is pressed, *unit setters* will appear on the screen under the arrow; it will erase when a new answer is entered. Notice that no -at- instruction is specified to control the display of *unit setter*; the display automatically begins at a preset position below the arrow.

CHECK YOUR UNDERSTANDING

Directions: If you wish to practice using the -helpop-, -dataop-, and -labop- instructions, use your file space to do so now.

4-E. DISPLAY CONTROL INSTRUCTIONS



Identify the effects of -pause-, -erase- with a tag, -inhibit- with an *erase* tag, -erase- with no tag, and -end- with a *lesson* tag.

Activities 4-A through 4-D present instructions that allow you to control the sequence of the units you code. This activity presents instructions that control or manipulate the displays within a unit. It discusses the purposes and formats of:

-pause-
-erase- (with a tag)
-inhibit- (with an *erase* tag)
-erase- (with no tag)

It also discusses the use of the -end- instruction to end a lesson.

THE -pause- INSTRUCTION

Use the -pause- instruction when you want to bring up parts of a unit display one at a time. For example, when each point you make is so important that you want the student to study it before going on to the next point, or when a full screen of text would overwhelm the student, making it necessary to bring up only portions at a time, -pause- can achieve the desired results.

Two forms of the -pause- instruction are:

```
pause    keys=LIST OF KEYS
```

keys that will cause execution to continue

```
pause    NUMBER OF SECONDS
```

The first format holds the pause until a key listed in the tag is pressed. For example

```
pause    keys=next,back
```

will halt execution until the student presses either the NEXT or the BACK key. The keys pressed will then perform their specific functions, as coded. If this code is used,

```
unit    two
back    one
next    three
at      3020
write   Press BACK to review.
        Press NEXT to continue.
at      1109
write   1. Read the case study.
pause   keys=next,back
at      1613
write   2. Decide what action should be taken.
pause   keys=next,back
at      1813
write   3. Enter your decision.
```

the student can press BACK, which will cause an immediate branch to *unit one*, as specified by the -back- instruction. If the student presses the NEXT key, the next display of text will appear. When NEXT is pressed upon completion of the unit, the student will begin *unit three*, as specified in the -next- instruction.

The second format of the -pause- instruction is used to specify a number of seconds. Execution will halt for the number of seconds specified. This allows you to pace the presentation; the student has no control over the pause. Thus, -pause- with a seconds tag could be used to pace the presentation of important points or to bring up letters or words one at a time.

Because students have different reading and comprehension rates, care should be taken when using -pause- with a seconds tag; students can be bored or frustrated by displays that are paced too slowly or too rapidly.

THE -erase- INSTRUCTION WITH A TAG

The -erase- instruction with a tag is used to erase specified portions of a display within a unit. Its format is:

```
erase   NUMBER OF CHARACTERS,NUMBER OF LINES
```

The first *argument* (part) of the tag specifies the number of *characters* to be erased. The second argument of the tag specifies the number of *lines* to be erased. For example, if you want to erase a *Press NEXT to continue* message located on line 30, beginning with the first character position, and a *Press BACK to review* message on the next line, you would use this instruction:

```
erase   26,2
```

The *26* is the number of characters in the longest line. The *2* is the number of lines to be erased.

If only one argument is used, such as:

```
erase    26
```

the instruction will cause the specified number of characters in one line to be erased, in this case, 26 characters.

When using -erase- with a tag, be sure to specify which lines are to be erased; use an -at- instruction. For example, if you want to erase the first 14 characters of line 2Ø, use this code.

```
at       2ØØ1
erase    14
```

THE -inhibit- INSTRUCTION

The -inhibit- instruction, when used with an *erase* tag, allows you to move to the next main unit in your sequence without erasing the current unit. If you want to change main units without a full screen erase, place an *inhibit erase* instruction in the unit you wish to retain when the change occurs. For example, if you want *unit one* to remain when the student begins *unit two*, place the instruction in *unit one*.

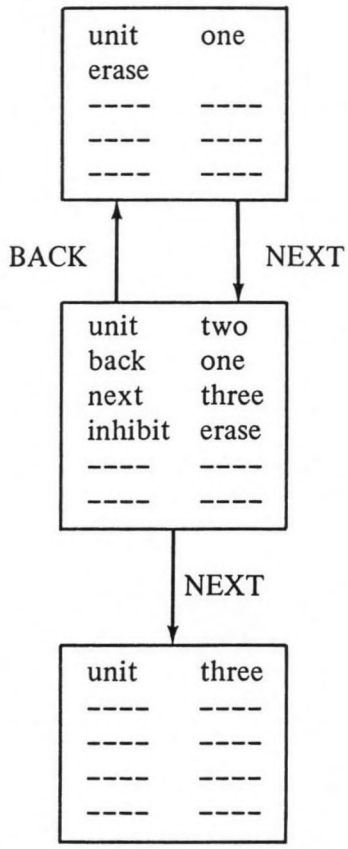
If -jump- is used to access the new main unit, place the *inhibit erase* instruction before the -jump- instruction. With this code

```
unit     two
back     one
next     three
at       5Ø5
write    How many weeks of vacation does a five-year
         employee receive?
arrow    8Ø5
answer   (three,3)
wrong    (two,2,four,4)
inhibit  erase
jump     newmain
```

The *inhibit erase* instruction is executed only if the student types in one of the specified wrong answers. The *inhibit erase* instruction only executes in the unit in which it is placed; if you wish to retain more than one unit on the screen, you must include the *inhibit erase* instruction in each unit you wish to retain.

THE -erase- INSTRUCTION WITH A BLANK TAG

The -erase- instruction with a blank tag will cause a full-screen erase. It is often used in cases in which an *inhibit erase* instruction has been included in one of the units of a sequence. For example, in this sequence,



the *inhibit erase* instruction will cause *unit two* to remain on the screen if the student enters *unit one* by pressing BACK, or *unit three* by pressing NEXT. The programmer wants this to occur when *unit three* begins; however, the programmer wishes to erase *unit two* if the BACK key is pressed. Thus, -erase- with a blank tag is included in *unit one*; the *unit two* display will erase when *unit one* is accessed.

THE -end- INSTRUCTION

The -end instruction has a special use; it can end a lesson. When used for this purpose, the -end- instruction looks like this:

```
end      lesson
```

This instruction must be placed in the last unit you want the student to see. Otherwise, the student will be shown auxiliary unit code, or any code you have included after the last unit in the lesson sequence.

CHECK YOUR UNDERSTANDING

Section 1

Directions: Answer the questions in the blanks provided.

1. What effect does this instruction

```
pause    keys=next,data
```

have on unit execution?

_____

2. What effect does this instruction

```
pause    2
```

have on unit execution?

_____

3a. What does the first argument of this erase instruction tag specify?

```
erase    10,3
```

_____

3b. What does the second argument specify?

_____

4. How many lines will this instruction erase?

```
erase    4
```

_____

5. Which instruction allows you to move to the next main unit in your sequence without erasing the current unit?

_____

6. Which instruction causes a full screen erase?

_____

7. Which instruction must be included in the last unit of a lesson?

_____

Answers

1. It halts execution until either the NEXT or DATA key is pressed.
2. It halts execution for two seconds.

3a. 1Ø characters

3b. 3 lines

4. 4 characters of one line
5. inhibit erase
6. -erase- with a blank tag
7. end lesson

Section 2

Directions: If you wish to practice using the instructions presented in this activity, use your file space to do so now.

4-F. COMPREHENSIVE PRACTICE



Given programmable-ready material, write a program using the instructions presented in this unit.

You've learned and practiced a number of instructions in this unit. This activity gives you the opportunity to put them all to work in a lesson.

The following pages contain programmable-ready material for a lesson (the flowchart, displays, and directions). It is up to you to determine the instructions you must use to attain the specified results. Study the material, and then code the lesson in your file space. Delete the code in any blocks you have used previously; this lesson requires several blocks. You will probably require two lesson parts of file space to complete this lesson. *Ask your learning center technician* to lengthen your file space to permit a two-part lesson.

Remember that when condensing your lesson to check it as a student, you can prevent specific blocks from condensing; press SHIFT-p and the letter of the block or blocks that should not be condensed. Then press NEXT. An additional time saver: To move between blocks quickly without going back to the block display, press SHIFT-NEXT to move to the next block; SHIFT-BACK will take you backward block by block.

IMPORTANT NOTE

The lesson you program in this activity will be used again in unit 6 and unit 7, where you will add to and change your code using the new instructions you learn. Therefore, it is very important that you:

1. Complete *all* coding for the lesson.
2. *Save* this lesson; do not delete any portion of it.
3. Prevent these blocks from condensing (SHIFT-p on the block display) when you are ready to go on to the next unit.

Unit 6 and unit 7 also provide the opportunity to practice using instructions before you code the final cumulative program; use empty blocks for these practice exercises. Delete this practice code when you are ready to program the final activity in each unit.

CHECKING YOUR CODE

If you have difficulty coding, or if you want to see if you have coded in the most efficient way, type *∅plato4* on the author mode display; *∅plato4* contains a complete, working program for the classification lesson.

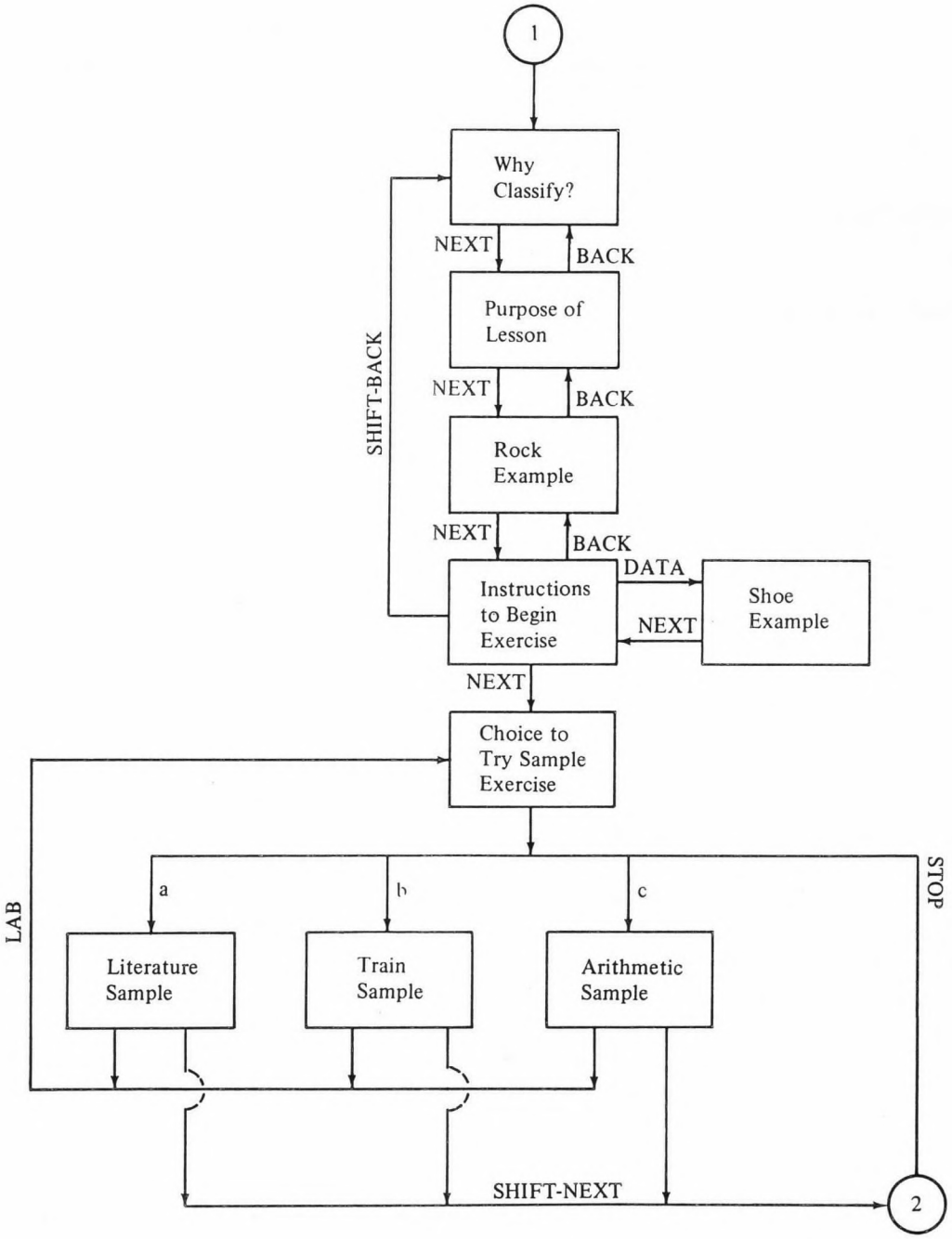
It also frequently comments on the instructions used (anything preceded by an asterisk or a dollar sign is a comment); these comments are usually explanations of why parts of the lesson are coded as they are, or explanations of what a particular instruction accomplishes.

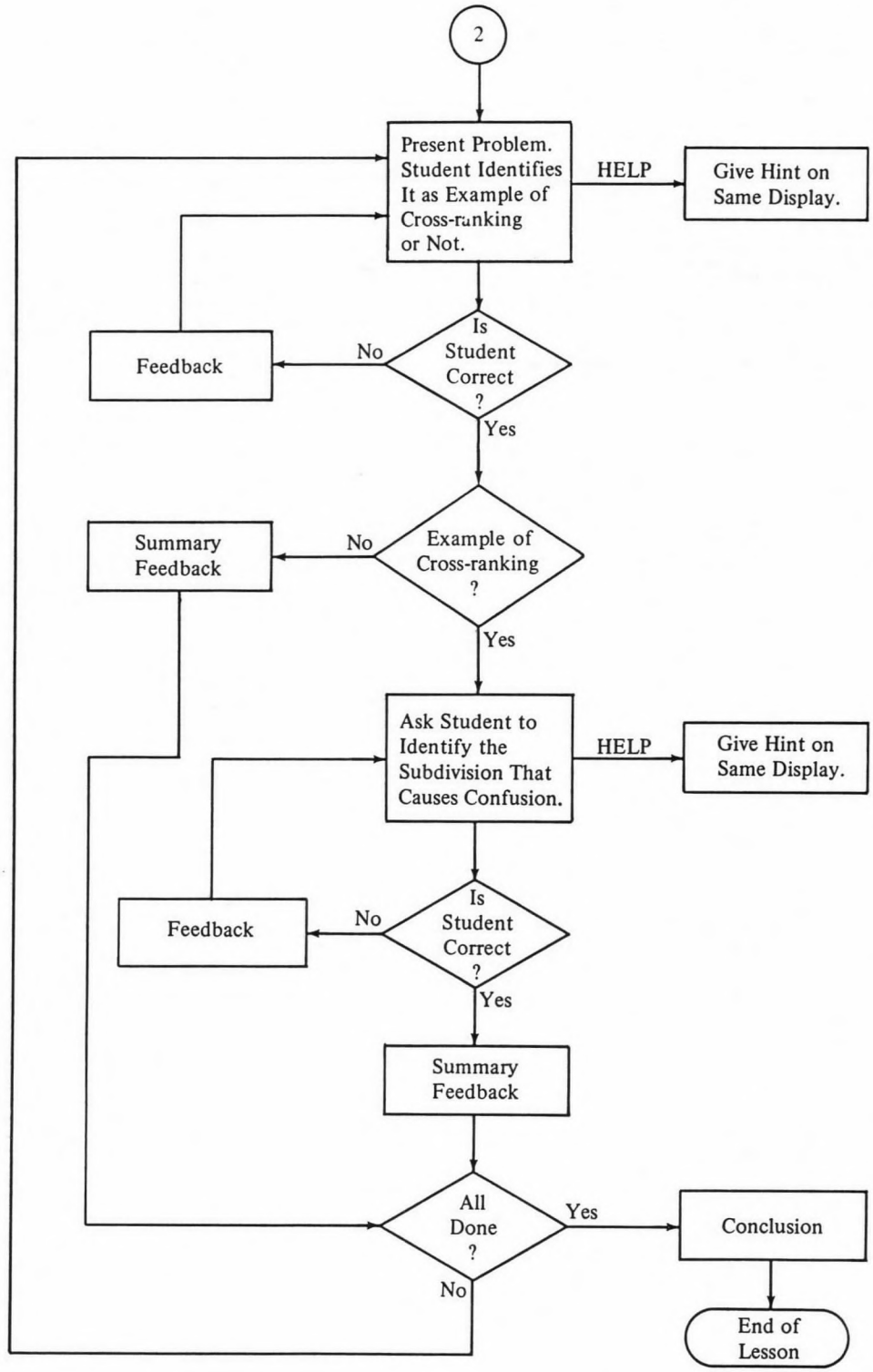
PROGRAMMABLE-READY MATERIAL

Goal Statement

This lesson explains the purposes of classification schemes, with emphasis on maintaining a single basis of division. Students will be asked to identify cases where cross-ranking occurs and to identify the inappropriate subdivisions in these cases.

Flowchart


1
Why Classify?
NEXT
BACK
Purpose of Lesson
NEXT
BACK
Rock Example
NEXT
BACK
SHIFT-BACK
Instructions to Begin Exercise
DATA
NEXT
Shoe Example
NEXT
Choice to Try Sample Exercise
a
b
c
STOP
LAB
Literature Sample
Train Sample
Arithmetic Sample
SHIFT-NEXT
2


2
Present Problem. Student Identifies It as Example of Cross-ranking or Not.
HELP
Give Hint on Same Display.
Is Student Correct ?
No
Feedback
Yes
Example of Cross-ranking ?
No
Summary Feedback
Yes
Ask Student to Identify the Subdivision That Causes Confusion.
HELP
Give Hint on Same Display.
Is Student Correct ?
No
Feedback
Yes
Summary Feedback
All Done ?
Yes
Conclusion
No
End of Lesson

Storybook

General Instructions to the Programmer

The displays are identified by phrases on the flowchart. Accompanying the flowchart are detailed documents that describe each display and give general instructions. For ease in identification, each document is headed with the same phrase that is found on the flowchart.

These symbols on the documents are used consistently to mean:



Break the display at this point. Pause, and write *Press NEXT* at the bottom of the display. Upon a NEXT key press, erase this phrase and continue to display the remaining text.



Notify the student that the NEXT key is active by placing a *Press NEXT* at the bottom of the display.



Notify the student that both the NEXT key and the BACK key are active by placing a *Press BACK* in the lower left-hand corner and *Press NEXT* in the lower right-hand corner of the display.



The circle and the number contained in the circle are not to appear on the display. However, there are special instructions for this item at the bottom of the document.



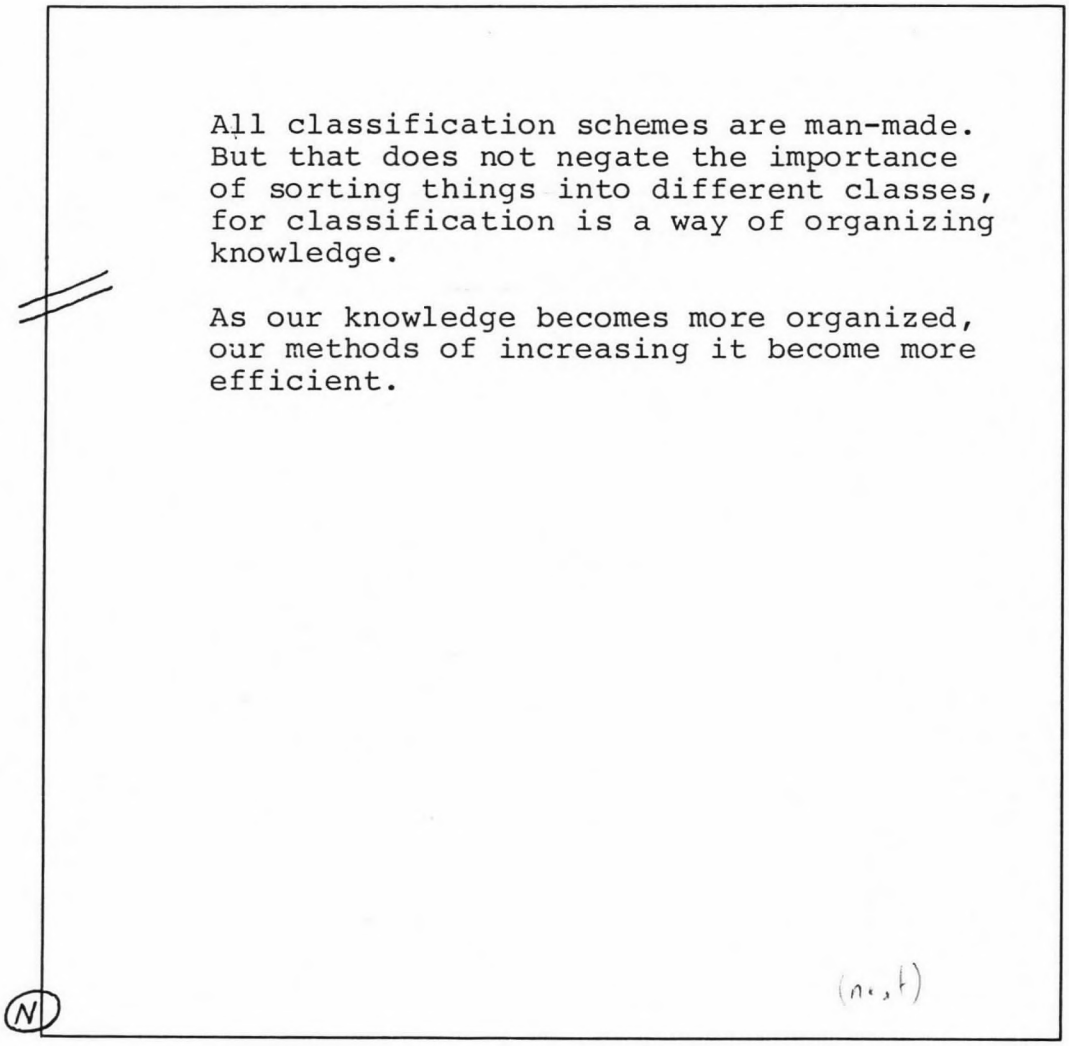
The brackets are not to appear on the display. However, this text is used repeatedly in a series of displays that all have the same format. You will find the text specific to each display of the series attached to the document upon which you find this symbol.



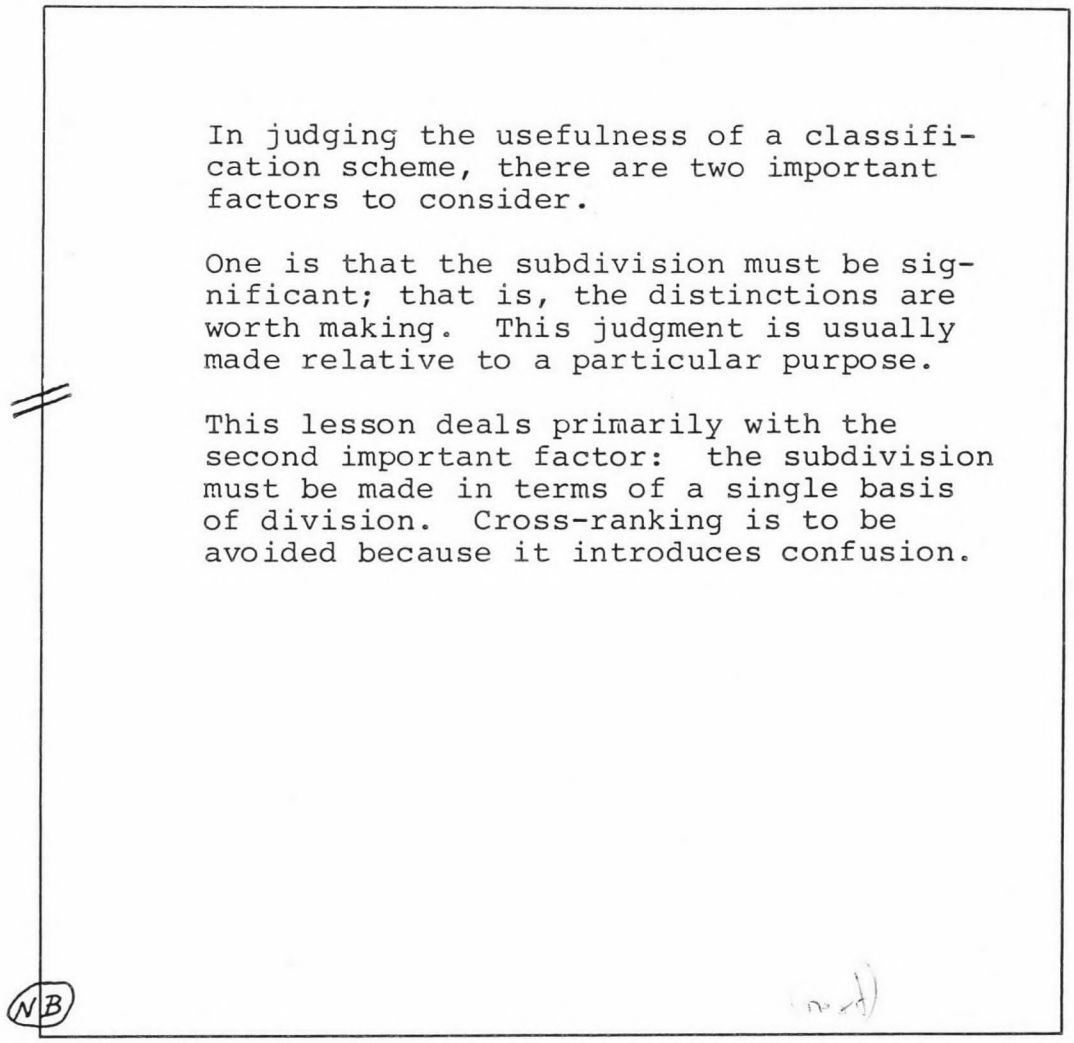
An arrow indicating that the student is to respond to the question should be displayed after the question is displayed.

Display Documents

Why classify?



Purpose of lesson



Rock example

For example, rocks may be classified as igneous, sedimentary, or metamorphic. This scheme may be valuable to a geologist involved in oil exploration.

Rocks may also be classified according to their hardness: a scheme valuable for industrial uses.

Or, they may be classified according to how much the stone bends light, which is called the index of refraction. This scheme would be valuable to a gem collector, because the refractive index affects the amount of sparkle.

But a classification such as this:

Very Hard, Sedimentary, Metamorphic

is inappropriate because it has introduced confusion by cross-ranking. It is true that most igneous rocks can be categorized as very hard, but it is not true for all igneous rocks. Therefore, some rocks would belong to two categories.

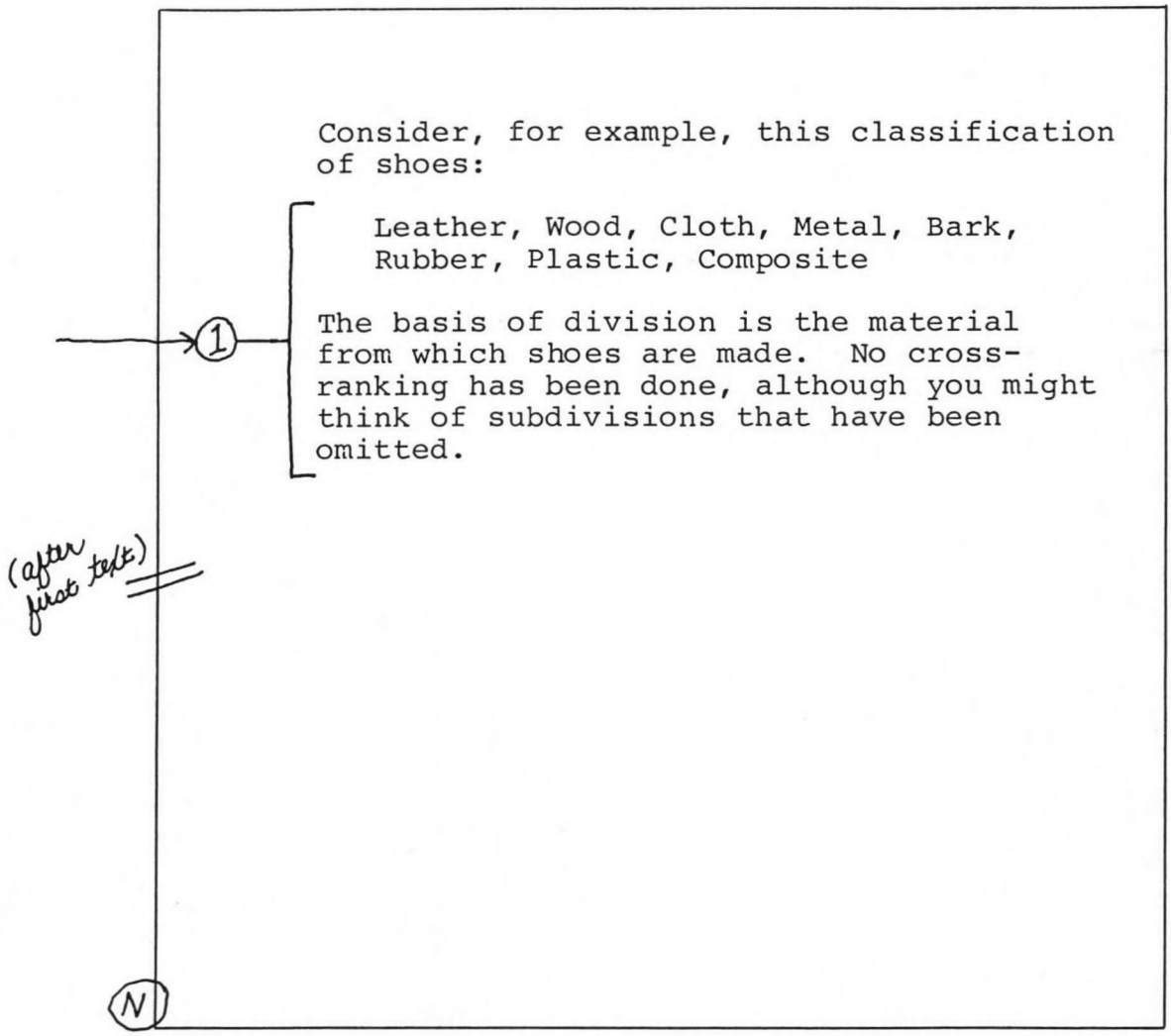


Instructions to begin exercise

In this lesson, you will be asked to identify instances where cross-ranking has occurred. In those cases where a classification is confusing because one subdivision causes cross-ranking, you will be asked to identify the subdivision that is inappropriate.

Press NEXT to begin the exercise
DATA if you need another example
BACK to review previous display
SHIFT-BACK to review the introduction

Shoe example

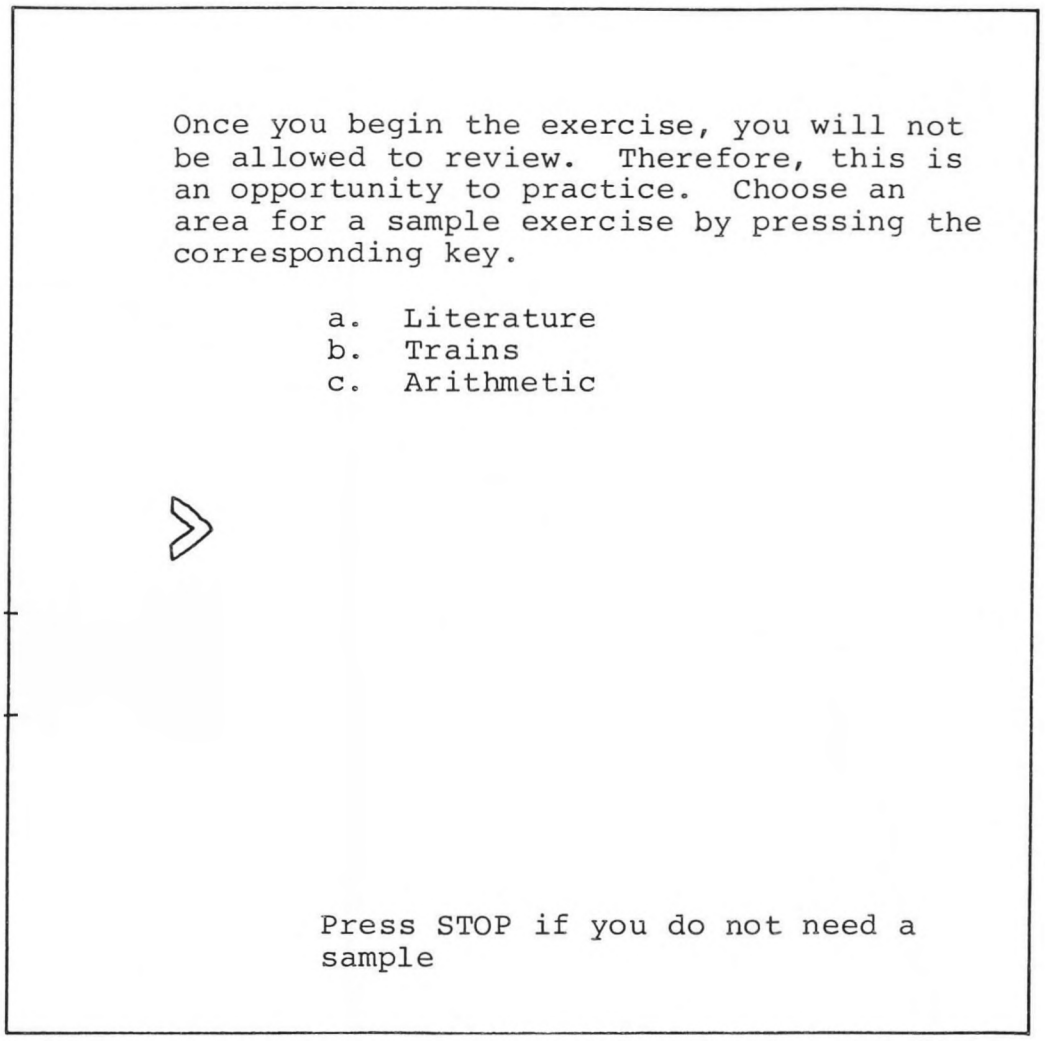


Special instructions: ① After the student presses the NEXT key, only the text marked by brackets should be erased. The following text should be presented in its place:

Leather, Wood, Cloth, Sport, Plastic, Rubber, Composite

Now a new subdivision, sport shoes, has been added, and an error of cross-ranking has been introduced. No longer is the basis of division single; there are two: material from which shoes are made and purpose for which shoes are used. The classification is not useful, because it is not clear-cut. Some shoes (like tennis shoes) may belong to two subdivisions.

Choice to try sample exercise

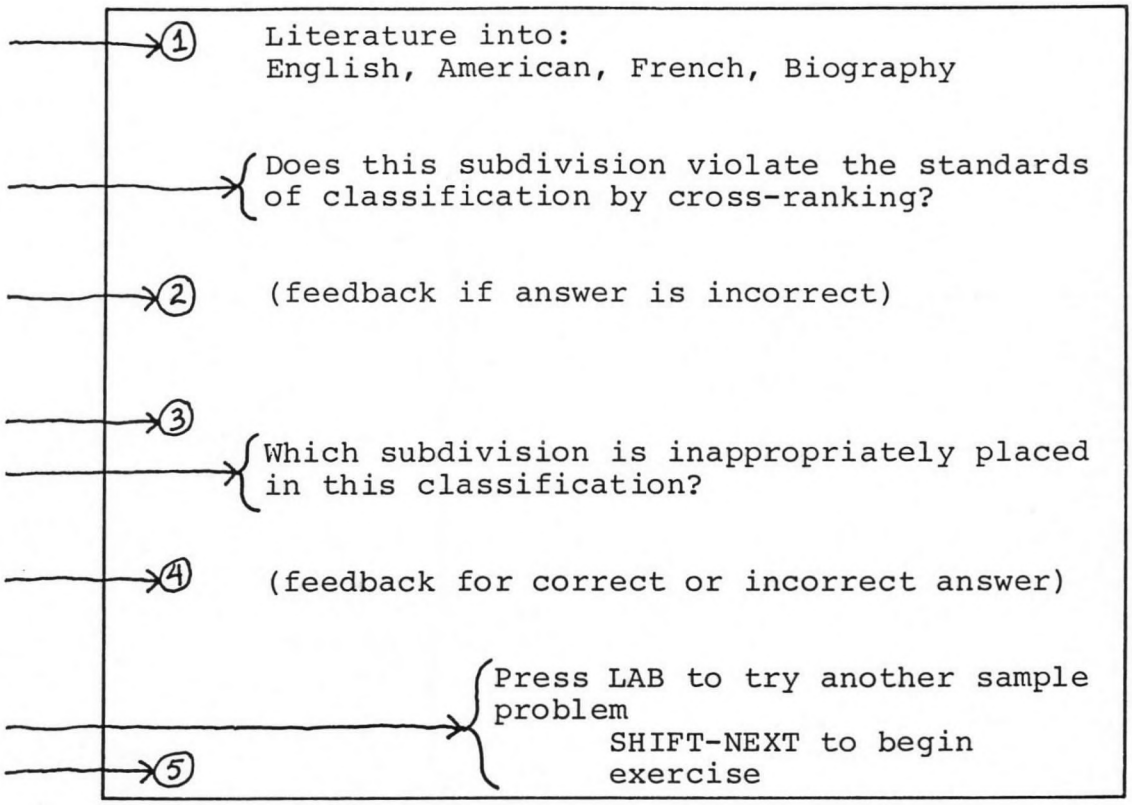


Special instructions: (1) If the student enters the response a, b, or c, go immediately to the appropriate sample exercise.

(2) If the student does not press one of these keys: a, b, c, or STOP, give this feedback and require one of those responses:

> Type a, b, or c, please. Or, you may press the STOP key if you do not want to try a sample exercise. Press NEXT and try again.

Literature sample (format for all three samples)



Special instructions: (1) Text for the remaining sample problems is attached.

(2) All three samples are cross-ranked. Therefore, the correct answer will be "yes" in every case. If the student answers "yes" or "y", a simple "ok" is sufficient. But, if the student answers "no" or "n", judgment should be "no" and the following feedback should be presented:

> Sorry, but one of the categories is cross-ranked. Press NEXT and try again.

The student should be required to enter the correct answer before proceeding.

(3) This question should be presented immediately after the student has responded "yes" to the previous question.

(4) The student should be required to answer correctly before proceeding. Correct answers and feedback for all answers (correct or incorrect) are attached.

(5) Turn on the option to begin the exercise after the student has correctly answered both questions. Add this line to the display at that time.

{ Bracketed text is the same in all three sample problems.

Attachment for literature sample and other samples

Text for the three samples:

Literature into:
English, American, French, Biography

Trains into:
Diesel, Electric, Freight, Steam

Fractions into:
Improper, Proper, Reduced

Sample	Correct Answer	Feedback if Correct	Feedback if Incorrect
Literature	biography	Yes, "biography" is a classification according to content; the others are classified by culture.	This categorizes literature by culture, as do all but one. Press NEXT and try again.
Trains	freight	Yes, "freight" is a classification according to use; the others are classified by engine type.	This categorizes trains by engine type, as do all but one. Press NEXT and try again.
Fractions	reduced	Yes, "reduced" is a classification that denotes the lack of a common divisor for the numerator and denominator; the others classify by relative size of the numerator and denominator.	This categorizes fractions by comparing the size of numerator and denominator as do all but one. Press NEXT and try again.

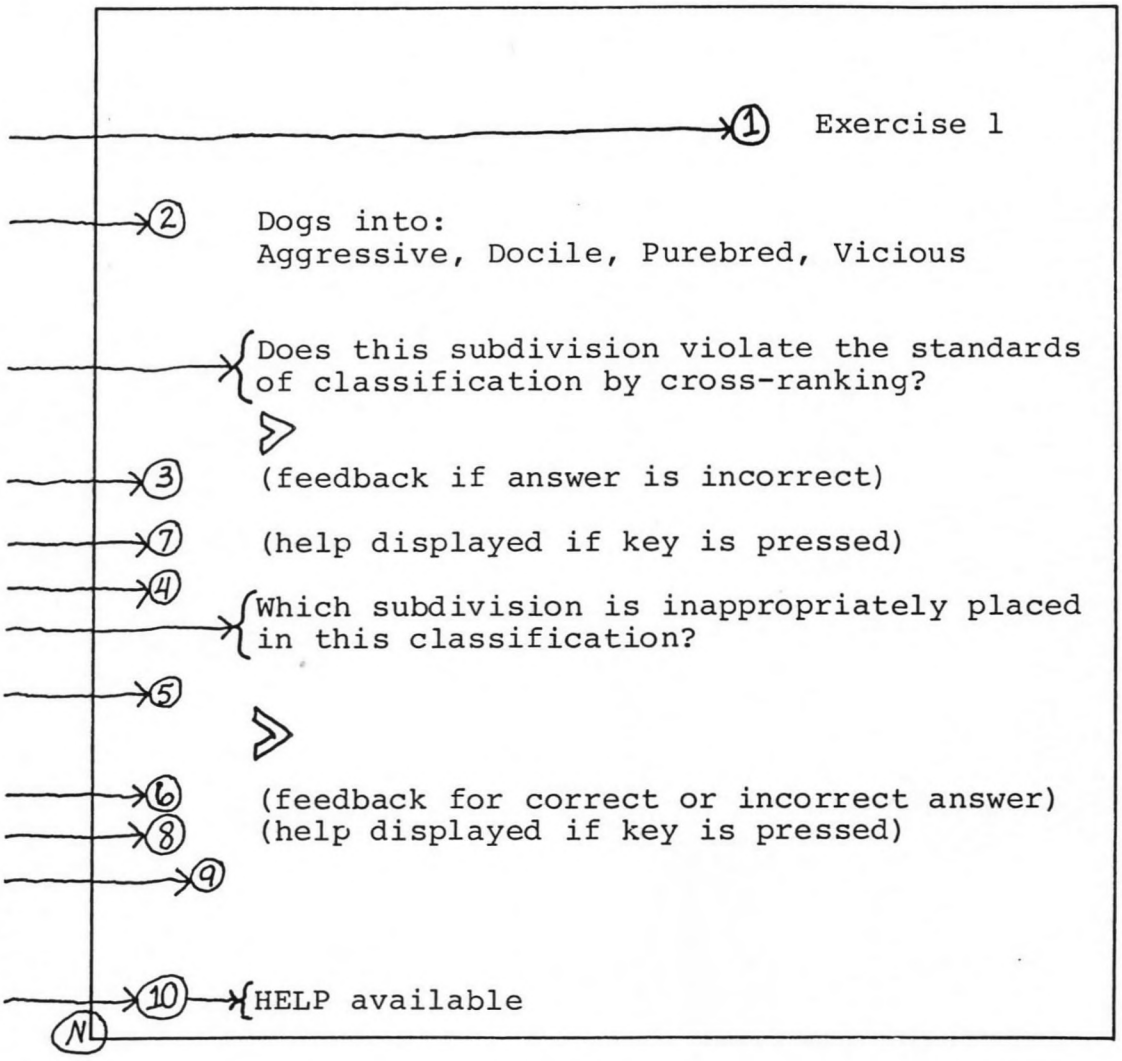
Special instructions: Accept reasonable misspellings and do not require capitalization.

Since the student is asked to choose from a list, do not provide feedback for unanticipated responses.

In each sample, the correct answer is listed. Incorrect anticipated responses for each sample are those remaining choices for the sample.

The student may add extra words to the response. The important thing is that the correct answer is included in the response.

Problem 1 (format for all problems)



Special Instructions: ① The number of the exercise will depend on which problem the student is presently working on.

② Text for each problem is attached to the specific problem.

③ No feedback is necessary if the student answers correctly (other than the standard "ok", that is). However, if the correct answer is "yes" and the student answers "no" or "n", provide this feedback:

> Sorry, but one of the categories is cross-ranked. Press NEXT and try again.

Continuation of special instructions for
problem 1 (format for all problems)

If the student answers "yes" or "y", and the correct answer is "no", provide this feedback:

> Sorry, but none of the categories is cross-ranked. Press NEXT and try again.

In either case, if the student answer is incorrect, require that the correct answer be entered.

⑦ ⑧ If the student presses the help key, display the appropriate help message for the specific problem without erasing what is already on the display. (Help messages will be found on the attached documents for specific problems.) Erase the help message before presenting any feedback or additional questions (same message in both instances).

④, ⑤, ⑥, and ⑧ are to be presented ONLY if the correct answer to the first question is "yes" (meaning that an instance of cross-ranking has occurred).

④ The student should be required to answer correctly before proceeding. Correct answers and feedback for all answers (correct or incorrect) are included on the documents attached for each specific problem.

⑨ is to be presented ONLY if the correct answer to the first question is "no" (meaning that no instance of cross-ranking has occurred). This is summary feedback for the specific problem, explaining why no instance of cross-ranking occurred and identifying the subdivision that was used throughout the problem.

⑩ Availability of HELP and the message indicating that HELP is available should be turned off when the student has completed the problem satisfactorily. HELP messages are included in the text attachment.

Attachment for problem # 1

Text to insert at (2):

Dogs into:
Aggressive, Docile, Purebred, Vicious

Correct answer for question: Does this subdivision violate . . . . . yes

---

Help to display at (7) or (8) if key is pressed:

Three of the subdivisions are classifications of temperament. The fourth subdivides by breeding.

Correct answer to question: Which subdivision is . . . . . (if answer to first question is "yes") purebred

---

Feedback to insert at (6) if student answers correctly:

Yes, "purebred" is a classification according to breed; the others are classified by temperament. Confusion would result if you attempted to classify Doberman Pincer (a purebred variety noted for viciousness).

Feedback to insert at (6) if student answers incorrectly:
(aggressive, docile, or vicious)

This categorizes dogs by temperament, as do all but one. Press NEXT and try again.

Summary feedback to insert at (9) if correct answer to first question is "no" (indicating that the second question will not be presented):

not applicable

Attachment for problem # 2

Text to insert at (2):

Art into:
Literature, Music, Painting, Sculpture

Correct answer for question: Does this subdivision violate . . . . . no

---

Help to display at (7) or (8) if key is pressed:

All of these subdivisions describe a medium through which artistic expression is made.

Correct answer to question: Which subdivision is . . . . . (if answer to first question is "yes") not applicable

---

Feedback to insert at (6) if student answers correctly:

not applicable

Feedback to insert at (6) if student answers incorrectly:

not applicable

Summary feedback to insert at (9) if correct answer to first question is "no" (indicating that the second question will not be presented):

Yes. All of these subdivisions describe a medium through which artistic expression is made. Cross-ranking is avoided.

Attachment for problem # 3

Text to insert at (2):

Canned goods into:
Fruits, Juices, Meats, Vegetables

Correct answer for question: Does this subdivision violate . . . . . yes

---

Help to display at (7) or (8) if key is pressed:

Three of these describe the variety of food. The fourth describes the form.

Correct answer to question: Which subdivision is . . . . . (if answer to first question is "yes") juices

---

Feedback to insert at (6) if student answers correctly:

Yes, "juices" is a classification according to form; the others are classified by variety. Confusion would be introduced if, for example, you attempted to classify apricot nectar.

Feedback to insert at (6) if student answers incorrectly:

This categorizes foods by variety, as do all but one. Press NEXT and try again.

Summary feedback to insert at (9) if correct answer to first question is "no" (indicating that the second question will not be presented):

not applicable

Attachment for problem # 4

Text to insert at (2):

Basic Energy Sources into:
Coal, Electricity, Oil, Solar

Correct answer for question: Does this subdivision violate . . . . . yes

---

Help to display at (7) or (8) if key is pressed:

Three of these represent energy forms that can be used in the natural state; the fourth represents a form that must be created from natural forms.

Correct answer to question: Which subdivision is . . . . . (if answer to first question is "yes") electricity

---

Feedback to insert at (6) if student answers correctly:

Yes, "electricity" introduces confusion by cross-ranking; the others are forms of energy usable in the natural state. Since usable electricity is derived from these forms, classification will become more difficult.

Feedback to insert at (6) if student answers incorrectly:
(coal, oil, or solar)

This categorizes energy sources by usability in the natural state, as do all but one. Press NEXT and try again.

Summary feedback to insert at (9) if correct answer to first question is "no" (indicating that the second question will not be presented):

not applicable

Attachment for problem # 5

Text to insert at (2):

Hats into:
Derby, Felt, Fur, Straw

Correct answer for question: Does this subdivision violate . . . . . yes

---

Help to display at (7) or (8) if key is pressed:

Three of these describe the fabric from which hats are made; the fourth describes the style.

Correct answer to question: Which subdivision is . . . . . (if answer to first question is "yes") derby

---

Feedback to insert at (6) if student answers correctly:

Yes, "derby" is a classification according to style; the others are classified by fabric.

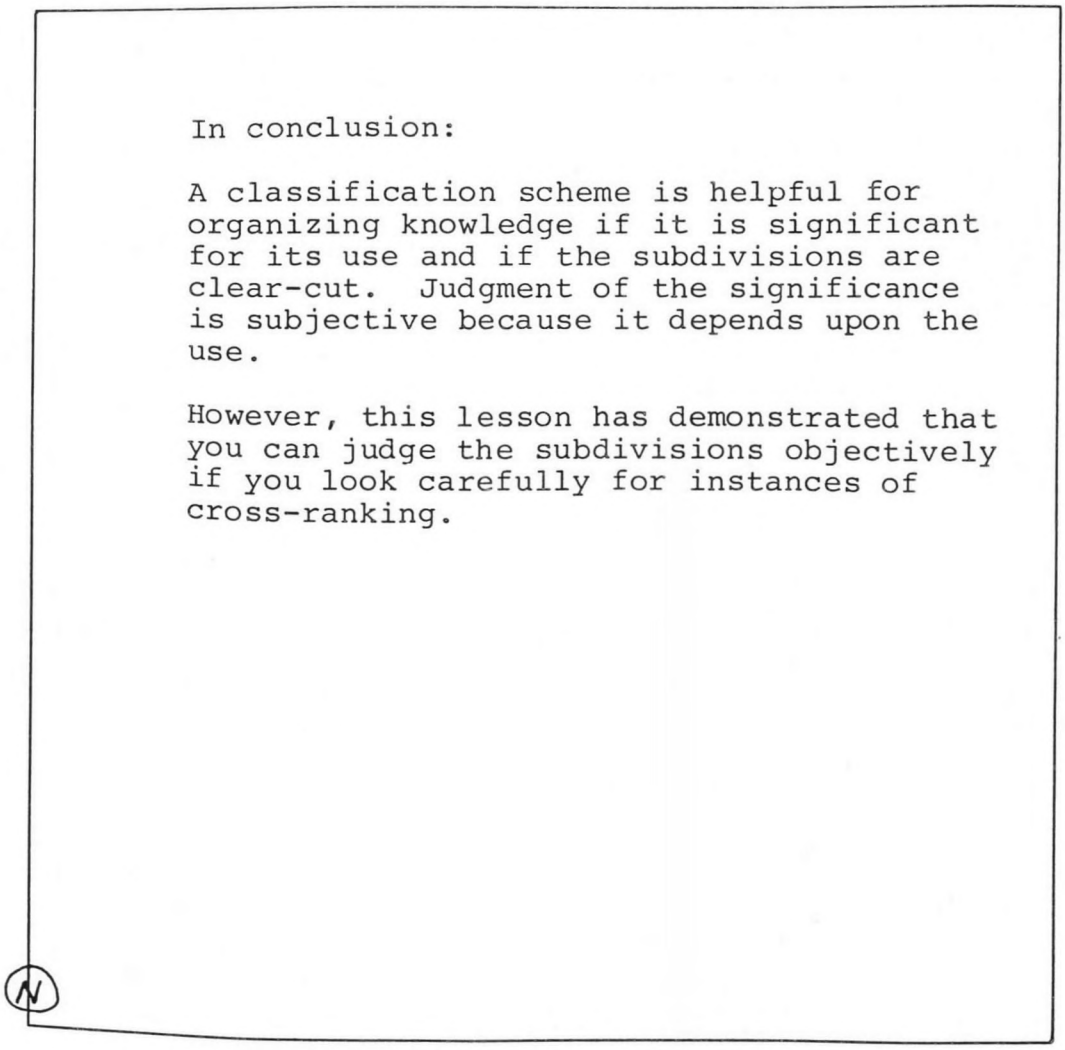
Feedback to insert at (6) if student answers incorrectly:
(fur, felt, or straw)

This categorizes hats according to the fabric from which they are made, as do all but one. Press NEXT and try again.

Summary feedback to insert at (9) if correct answer to first question is "no" (indicating that the second question will not be presented):

not applicable

Conclusion


In conclusion:
A classification scheme is helpful for organizing knowledge if it is significant for its use and if the subdivisions are clear-cut. Judgment of the significance is subjective because it depends upon the use.
However, this lesson has demonstrated that you can judge the subdivisions objectively if you look carefully for instances of cross-ranking.
N

5

Information and Communication

As you begin to experiment with the PLATO Author Language, you will require information unique to your situation. This unit presents several methods of obtaining this information, including the *Control Data PLATO Author Language Reference Manual* and the on-line *aids* lesson. It also presents several communication options that will enable you to discuss your problems with other system users.

INSTRUCTIONAL OBJECTIVES

At the end of this unit, you should be able to:

- Identify procedures for using the *Control Data PLATO Author Language Reference Manual*
- Identify procedures for accessing the *aids* lesson to obtain information
- Identify procedures for using the TERM-talk option to communicate with other system users
- Identify procedures for putting yourself on the users list
- Identify procedures for turning your talk options on and off
- Identify procedures for using the TERM-consult option to contact system consultants
- Identify procedures for reading, writing, and sending notes via the PLATO system

LEARNING ACTIVITIES

_____ 5-A. The *Control Data PLATO Author Language Reference Manual.* Text/Reference Reading: In this activity, you will use the *Control Data PLATO Author Language Reference Manual* to locate information on instructions.

_____ 5-B. The On-line Reference: *Aids*. Text/CAI: This activity discusses the access and use of the *aids* lesson, an on-line reference manual.

_____ 5-C. The TERM-talk Option. CAI: This activity shows you how to communicate with other users on the system through the TERM-talk option.

_____ 5-D. The Users List and Talk Options. Text/CAI: In this activity, you will place yourself on the system users list and turn on your talk options.

_____ 5-E. The TERM-consult Option. CAI: This activity shows you how to contact system consultants using the TERM-consult option.

_____ 5-F. PLATO Notes. CAI: This activity shows you how to write and send notes to other system users via the PLATO system.

TESTING DIRECTIONS

Now sign on to a Control Data PLATO terminal. If you are very familiar with the subject of obtaining information and communicating via the PLATO system, you may wish to test out of some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.



5-A. THE *CONTROL DATA PLATO AUTHOR LANGUAGE REFERENCE MANUAL*

Identify procedures for using the *Control Data PLATO Author Language Reference Manual.*

Several manuals about the PLATO system have been published by Control Data Corporation. One in particular, the *Control Data PLATO Author Language Reference Manual*, is especially helpful in explaining instructions. In this activity, you will use this reference manual to locate information about the instructions you learned in previous units of this course. You will also be given a handy pocket reference manual containing instruction formats for many PLATO Author Language instructions.

USING THE REFERENCE MANUAL

Obtain the *Control Data PLATO Author Language Reference Manual* from your learning center technician. Turn to the contents page. Notice that each instruction is classified according to type. Because the -at- and -write- instructions are display instructions, they are listed under *Display Instructions*. The -unit- instruction is a lesson control instruction. The -arrow- instruction is a response handling instruction, as are -answer-, -wrong-, -specs-, -no-, and -endarrow-. All other instructions are listed under their appropriate categories. The pages listed in the contents contain brief descriptions of the instructions.

Now turn to the table that begins on the first page in section five. The table lists *two* page references for each PLATO Author Language instruction. One is the short description of the instruction listed on the contents page; it is used mainly to give the possible instruction formats. The other page referenced contains a longer, more detailed description of the instruction and its use.

REFERENCE READING INSTRUCTIONS

Now use the contents pages and the table listings in section five to locate and read references to any or all of the instructions you have learned thus far in this course. Do not be concerned if you cannot understand the more detailed descriptions; the purpose of this activity is to learn to locate information in the manual, not to learn more about specific instructions. Also, you may notice that occasionally, the reference manual is not totally up-to-date. Although it is revised regularly, system changes may make portions of the manual you have in hand obsolete. Activity 5-B discusses an on-line reference manual called *aids*, which is updated immediately when the system is changed.

USING THE POCKET REFERENCE MANUAL

If you have not already received it, obtain a copy of the *Control Data PLATO Author Language Instruction Formats* from your learning center technician. This handy reference will help you if you should forget the format of any of the numerous instructions presented in this course.

SUMMARY

The contents page of the *Control Data PLATO Author Language Reference Manual* lists pages giving brief descriptions of each instruction. The table in the beginning of section five lists two page references for each instruction. The first is brief and the second is a more lengthy explanation. The *Control Data PLATO Author Language Instruction Formats* manual contains the general formats for many PLATO Author Language instructions.

5-B. THE ON-LINE REFERENCE: *AIDS*



Identify procedures for accessing the *aids* lesson to obtain information.

Although it is a fairly simple computer language, the PLATO Author Language is extensive and, in some ways, complex. This can cause problems for the author who wants specific information about it. Activity 5-A discusses the *Control Data PLATO Author Language Reference Manual*. This activity discusses another source of specific information, the on-line reference called *aids*.

The *aids* lesson contains two types of information: information about specific PLATO Author Language instructions and how they work, and information about general PLATO Author Language and system topics, such as response judging. Because it is actually a set of PLATO lessons, *aids* can be modified easily when the system changes.

This activity explains how to access *aids* and how to obtain information when you are in it. It also explains the function keys available in the *aids* lesson.

ACCESSING *AIDS*

Aids can be accessed from the author mode display and from the editor. To access *aids* from the author mode display, either enter the word *aids* and press the DATA key, or simply press *A*. If the latter method is used, it is necessary to press the DATA key only if you are not taken directly to the *aids* title display. To access *aids* from the editor, press *Q* and the NEXT key. You will be taken to the *aids* title display.

GETTING INFORMATION

When you are in *aids* and see the title display, you can obtain information by pressing either the NEXT key or the DATA key.

The NEXT Key

Pressing NEXT on the *aids* title display takes you to the first part of a two-part index. Here, you can either type the letter of the topic you wish to see, or you can press NEXT to go to the second part of the index. Pressing NEXT while in

the second part of the index returns you to the first part of the index. The information available from the index is generally either overview information about a topic or a list of specific items.

The DATA Key

If you want specific information about a topic, press DATA. This option, available throughout the lesson, takes you to a display where you will give the key word or phrase that specifies the information you desire. Enter the word or phrase and press NEXT.

Sometimes *aids* may fail to get you the information you want. Failure occurs when:

- The information is not contained in *aids*
- The PLATO system cannot interpret the word or phrase you entered
- The PLATO system misinterprets the word or phrase

If *aids* cannot interpret your request, a message will appear telling you so. This shows that the lesson authors did not anticipate anyone asking for information using that key word or phrase. You can then press NEXT and try a different key phrase.

If, after entering a key word or phrase, you are taken to a display that contains information other than that which you desired, a misinterpretation has occurred. This misinterpretation occurs if:

1. The authors did not anticipate the request.
2. The response judging mechanism of the PLATO system, which attempts to make the best possible match between the request and the mechanism's specified list, matches your response to the wrong topic.

In this case, press DATA and enter a new word or phrase.

FUNCTION KEYS

Several function keys are always available while you are in *aids*. The DATA key has already been mentioned, as has the NEXT key. Both can be used on the title display. Other keys available when you are actually in the lesson are NEXT, BACK, HELP, SHIFT-NEXT, and SHIFT-BACK. The LAB key is sometimes available.

The NEXT and BACK keys function as they usually do in a lesson. NEXT takes you to the next display, and BACK takes you to the previous display.

The HELP key takes you to a help sequence that explains partially how *aids* works and lists some of the contributors, as well as the most recent changes and additions. Once you are familiar with *aids*, your use of the HELP display will probably be limited. However, in case of confusion, this key is always available.

A SHIFT-BACK key press returns you either to a mini-index concerning a topic on the PLATO Author Language, or to the main index, which can also be accessed by pressing NEXT on the title display.

A SHIFT-NEXT key press takes you out of *aids* and returns you to the point from which you originally accessed *aids*. SHIFT-NEXT is used primarily when *aids* is accessed from the editor; it allows you to continue to edit where you left off. However, if you accessed *aids* from the author mode display, it returns you there. The only way to leave *aids* other than by pressing SHIFT-NEXT is to press SHIFT-STOP. The SHIFT-STOP key press has the same effect in all lessons: It immediately terminates the lesson.

The LAB key is sometimes available. It can serve various functions, but the most common is to give more detail about an instruction or to provide a simulation of instruction operation. If the LAB key is available, a message to that effect appears on the screen.

CHECK YOUR UNDERSTANDING

1. Go to the PLATO terminal. Using your author sign-on, access *aids*. Press the DATA key and type the word *write*. Press NEXT. Press NEXT until you have read the entire discussion. Press SHIFT-BACK to return to the index.
2. Press DATA and type *at* to locate information on the PLATO Author Language instruction -at- and look at some examples.
3. Experiment with *aids*. Look up the instructions you learned previously. When you have finished, press SHIFT-NEXT to return to the author mode display.
4. Sign off the system by pressing SHIFT-STOP until the *Press NEXT to begin* message appears.

SUMMARY

The following chart summarizes the keys used to access information through lesson *aids*:

Keys	*Where Keys Are Pressed*	*Where They Take You*
aids DATA	Author Mode Display	Lesson *aids*
A	Author Mode Display	Lesson *aids*
NEXT	*aids* title display	Two-part index
Q, NEXT	Editor	Lesson *aids*
DATA	Anywhere in *aids*	Display on which key words or phrases are entered
Keywords and phrases	Display accessed by pressing DATA	Specific information
NEXT	In index	Next page of index
NEXT	In lesson	Next display
BACK	In lesson	Previous display
HELP	Anywhere in lesson	Explanation of how *aids* works
SHIFT-BACK	In lesson	Mini-index or main index
SHIFT-NEXT	In lesson	To the point at which you accessed *aids*
SHIFT-STOP	Anywhere in lesson	Author mode display
LAB	Displays on which LAB is available	Instruction

5-C. THE TERM-TALK OPTION



Identify procedures for using the TERM-talk option to communicate with other system users.

While authoring on the PLATO system, you may encounter problems you wish to discuss with another author. The TERM-talk option allows you to use a PLATO terminal to communicate with any other user on the system.

CAI INTRODUCTION

This CAI activity explains how to use the TERM-talk option. The lesson references two terms with which you may be unfamiliar; the *users list* and *personal notes*. These topics are discussed in more detail in subsequent activities.

Now sign on to a Control Data PLATO terminal using your *student* sign-on. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "An Introduction to TERM-talk," press LAB to take the lesson. When you complete this lesson, read the summary in your text.

SUMMARY

The TERM-talk option allows you to communicate with any other user on the system. To use this option:

1. Press the TERM key.
2. Enter the word *talk* (type *talk* and press NEXT).
3. Type the sign-on name of the person with whom you wish to communicate, and press NEXT.
4. Type that person's group name and press NEXT.

If the message " _____ of group _____ not present" appears on the screen, it can mean:

a. The name or group is not typed correctly; in this case, press the TERM key again and retype the information.

b. No such sign-on exists.

c. The person you are paging is not signed on at this time.

If you receive the message "You cannot talk to _____ " it can mean:

a. The person is not an author.

b. The person's talk options are not turned on. (You will learn to turn on your talk option in the next activity.)

If you do not want to wait for a response, you can discontinue the paging by pressing BACK. If you do not wish to talk when paged, you can ignore the page, or press the TERM key and type *reject*. You can then type a message and send it with a NEXT keypress. Otherwise, the word *busy* will appear on the sender's terminal screen.

To respond to a page, press the TERM key and enter the word *talk*. When you press NEXT, the line you are typing erases and you can begin a new line at the arrow. End the talk by pressing SHIFT-BACK.

You can only talk to one person at a time; if the person with whom you wish to talk is already talking to someone else, a message tells you this. The person receives a message saying that you want to talk, and can page you later.

5-D. THE USERS LIST AND TALK OPTIONS



Identify procedures for putting yourself on the users list.

Identify procedures for turning your talk options on and off.

In order to locate users on the system to whom you can talk, you must access the users list. In order for people to talk to *you*, you must place your sign-on on the users list and turn on your talk options. This activity provides you with the opportunity to perform these actions.

Now sign on to a PLATO terminal. To see the users list:

1. Type *user* and press NEXT on the author mode display.

or

2. Press SHIFT-u on the author mode display.

You will see this display:

```
                Total users on system = 199


             Enter:  a physical site number,
                     a group name,
                     a user's name/group,
                or   a site-station number.


             >


Or press:  NEXT     (alone) to see all sites
           DATA     to see users at your logical site
           LAB      to display records/talk flag information

      SHIFT-DATA    to see a list of operations personnel.

                        HELP available
```

3. Press NEXT to see current users. Press NEXT again to see more users, until the list starts over.
4. Press BACK to get back to the author mode display.

To put yourself on the users list and turn on your talk options:

1. Press SHIFT-i on the author mode display. You will then see this display:

```
1.  Allow anyone to talk to you                        yes
2.  Allow only your account to talk to you             yes
3.  Allow only your group to talk to you               yes
4.  Current setting of TERM-busy flag                  not busy

5.  Allow anyone to send -ext-'s to you                no
6.  Allow only your group to send -ext-'s to you       no

7.  Current language                                   English
8.  Appear in "on-system" list of users                no

      Type the number to change a setting >
```

```
      Statistics for you / yours / minnb:
      Last use:   10/24/79  at  16.29.58.

  Session statistics:
      Hours on system.........     0.0
      CPU usage in TIPS.......     2.0
      Disk accesses in DAPM...     3.6

  Cumulative statistics:
      Total hours on system...     0.0
      CPU usage in TIPS.......     5.2
      Disk accesses in DAPM...    60.5
      Days on.................       1
      Sessions on.............       1

            HELP is available
```

2. You are concerned with numbers 1, 2, and 8. Type in these numbers one at a time. The *no* to the right of each option will change to *yes* for each number you press. You are now on the users list, and your talk options are on.
3. Press BACK to get to the author mode display. If you access the users list at this time, you will see your sign-on listed as one of the users.

5-E. THE TERM-CONSULT OPTION



Identify procedures for using the TERM-consult option to contact system consultants.

There may be times when none of the reference material available to you can solve your authoring problems. The PLATO system has consultants you may contact, using the TERM-consult option, when you need assistance.

CAI INTRODUCTION

This CAI activity explains the TERM-consult option and discusses the role of the consultant. Now sign on to a PLATO terminal using your *student* sign-on. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "TERM-consult," press LAB to take the lesson. When you finish this lesson, read the summary in your text.

SUMMARY

You can reach a consultant from almost anywhere on the system. To do so:

1. Press the TERM key.
2. Enter the word *consult*. The message, *A consultant has been notified of your request and will contact you shortly* will appear. You can continue with what you were doing while you wait.
3. When the consultant contacts you, go to the problem area or replot the display so the consultant sees what you see.
4. Press the TERM key to talk.
5. Press NEXT at the end of a line to leave talk mode, after which you may go to a new display.
6. If your message is more than one line in length, press LAB at the end of a line.
7. Type *goodbye* to end the consultation. An *end of consult* message will appear.

If you tire of waiting for a consultant:

1. Press the TERM key, type *consult*, and press NEXT.
2. Press SHIFT-HELP to cancel your request.

When no consultants are on the system, a message will tell you this. You can either wait for a consultant or try to get help somewhere else. If your message is logged, however, consultants will see it when they sign on.

5-F. PLATO NOTES



Identify procedures for reading, writing, and sending notes via the PLATO system.

When you wish to send a lengthy message to a person, or when the person with whom you wish to communicate is not available for TERM-talk, you can send a note using the notes file. The notes file also provides a source of information regarding the current status of the PLATO system. It is a cataloged collection of questions, answers, and comments about the system.

CAI INTRODUCTION

This activity shows you how to access the notes file. It also shows you how to read, write, and send notes. Now sign on to a PLATO terminal using your *student* sign-on. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "An Introduction to NOTES," press LAB to take the lesson. When you finish this lesson, read the following summary.

SUMMARY

The notes facilities allow you to send a note to one user or a group of users. All notes have three elements:

- The note header, which contains the note number and name of the author, and the time the note was written
- The body, which contains the message
- A list of available options, which lists keys to press

Personal notes are sent to a specific person. To write a personal note, press SHIFT-p on the author mode display. To read notes sent to you, press LAB. To write notes after reading a note, press BACK. Then type in the name and course of the person to whom you wish to send the note. Type the note when the arrow appears on the screen, pressing NEXT to start new lines. Press BACK to leave insert mode, SHIFT-NEXT to send the note, or SHIFT-BACK to exit without sending a note.

Size restrictions limit the number of lines per note to 2Ø, and the words per note to 12Ø. You can write an additional note if you reach your limit.

The message PERSONAL NOTES appears on the author mode display whenever someone sends you a note. SHIFT-p will take you to the note.

Group notes files contain notes accessible to a specific group of users. You must be a member of the group to read the notes. Access types for group notes include:

- *Director*, controls the notes file
- *Read-write*, can read or write any note
- *Write only*, can write notes but can't read notes
- *Read-respond*, can read notes and write responses to existing notes
- *Read only*, can only read notes
- *None*, can neither read nor write notes

To access a group notes file, type the name of the file on the author mode display and press NEXT. You can then read a note by typing the number of the note and pressing NEXT. You can write a note by pressing SHIFT-LAB.

Six other notes files are:

- *announce*, which is available to all PLATO authors to read announcements made by the PLATO system staff
- *helpnotes*, which is available to all PLATO authors to read and write notes about any subject
- *pbnotes*, which is available to all PLATO authors to read and write notes about any PLATO topic
- *plmnotes*, which is available to all PLATO authors to read and write notes about the PLM system
- *psonotes*, which is available to all PLATO authors to write notes to PLATO services consultants
- *rsm*, which is available to all PLATO authors to read and write about Requests for System Modifications

You can also access personal notes by pressing SHIFT-n on the author mode display. You will see a display on which:

- *a* takes you to announce
- *b* takes you to pbnotes
- *c* takes you to other group notes
- *d* takes you to personal notes

6

Variables, Calculations, and Conditional Instructions

The instructions that you have learned up to this point are all "unconditional" in the sense that their effects are not dependent or conditional on anything. This unit discusses conditional instructions—instructions that can produce different effects depending on the values of variables. Since the use of conditional instructions involves the use of variables, the unit begins with a discussion of variables and the calculation instructions used to define variables.

INSTRUCTIONAL OBJECTIVES

After completing this unit, you should be able to:

- Identify characteristics of variables
- Identify characteristics of PLATO variables: programmer-defined variables and system variables
- Identify the effects of instructions used to assign values to variables, assign names to variables, and display values of variables
- Identify the effects of random number generation instructions
- Identify the effects of -writec-, -answerc-, -wrongc-, and -calcs-
- Identify the effects of conditional forms of -next-, -back-, -stop-, -jump-, -do-, -join-, and -goto-
- Identify the effects of conditional instructions that branch within units
- Identify the effects of instructions used in pause processing
- Given programmable-ready material, write a program using variables, calculations, and conditional instructions

LEARNING ACTIVITIES

_____ 6-A. Variables. Text/CAI/Exercise: This activity defines the term *variable.*

_____ 6-B. PLATO Variables. Text/Exercise: This activity discusses two types of PLATO variables: programmer-defined variables and system variables.

_____ 6-C. Variables: Assigning Values, Assigning Names, and Displaying. Text/Exercise/CAI: This activity discusses instructions used to assign values and names to variables and to display values of variables.

_____ 6-D. Random Number Generation. Text/Exercise/CAI: This activity discusses two random number generation instructions: -randu- and -randp-.

_____ 6-E. Conditional Instructions: Writing, Judging, and Calculating. Text/Exercise/CAI: Conditional instructions are introduced; instructions for conditional writing, judging, and calculating are discussed in detail.

_____ 6-F. Conditional Instructions: Branching Between Units. Text/Exercise/CAI. Conditional forms of instructions that provide for branching between units are discussed.

_____ 6-G. Conditional Instructions: Branching Within Units. Text/Exercise/CAI: This activity describes conditional instructions that provide branching within units.

_____ 6-H. Pause Processing. Text/Exercise/CAI: The -pause- instruction is reviewed and the use of -pause- with the -keytype- and -group- instructions is discussed.

_____ 6-I. Programming with Variables, Calculations, and Conditional Instructions. Text/Exercise/CAI: Programmable-ready material requiring the use of variables, calculations, and conditional instructions is presented.

TESTING DIRECTIONS

Now sign on to a Control Data PLATO terminal. If you are very familiar with variables, calculations, and conditional instructions, you may wish to test out of some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.

6-A. VARIABLES



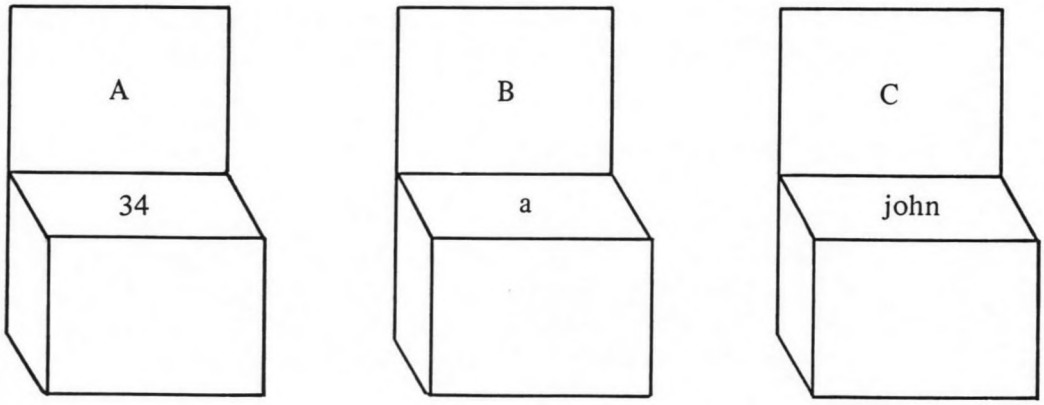
Identify characteristics of variables.

Suppose that you are writing a program in which you need to write a message on a lesson page. What would you do? You would use an -at- instruction followed by a -write- instruction in the appropriate unit. Now suppose that you need to write one of several messages on a lesson page. For example, say that on the student's first failure to answer a question you want to write *Try again*, on the second failure you want to give a hint, and on the third failure you want to write the correct answer. How would you do this? The solution is simple, but it requires a new instruction, the conditional write (-writec-).

To use -writec- and all other conditional instructions, however, you must be able to use *variables*. This activity defines the term *variable*. Activity 6-B defines two different types of PLATO variables. Activities 6-C and 6-D discuss instructions that allow you to use variables. The remainder of the unit presents conditional instructions, such as -writec-, that involve the use of variables.

WHAT ARE VARIABLES?

To a computer programmer, a variable is a location where things can be stored in the computer's memory. Think of a variable as a box. Just as you can store things in boxes, you can store things in variables. The things you store in variables (the contents of the box) are numbers, letters, and words. Look at the variables (boxes) shown: Variable A contains a number, variable B contains a letter; variable C contains a word.



The contents of a variable are called its *value*. The value of variable *A* is 34; the value of variable *B* is a; the value of variable *C* is john.

These boxes are called variables because their values (contents) can change (vary). You can, for example, change the contents of variable *A* from 34 to 35, a, john, or any other number, letter, or word.

THE NEED FOR VARIABLES

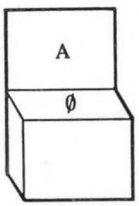
In most of the programs you will be writing, you will need to store things. You may, for example, want to store the student's name. Or you may want to store the number of correct responses in a quiz or the number of a student's attempts to answer a question. If you need to store something, you need a variable.

Perhaps the most common need for variables in programming is to store information that will be used later to make a decision. At the beginning of this activity, the problem was posed of how to give different feedback on successive failures to answer a question. The problem is to decide which feedback is to be presented. In order to make this decision, you must know the number of failures to answer the question. To know the number of failures, you must store the number in a variable.

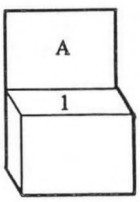
The flowchart in figure 12 is a concrete example of how the decision might be made.

Assume that the student's first three answers are wrong. Flowchart steps are listed in order on the left side of the page. The value of the variable at each step is shown on the right side of the page.

1. Start
2. Store ∅ in Variable A

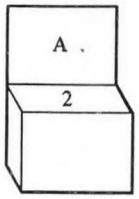


3. Ask Question
4. Student Answers
5. Add 1 to the Value in the Variable

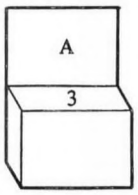


6. Response is wrong
7. Value of Variable A is 1

8. Write *Try again*



9. Student Answers
10. Add 1 to the Value in the Variable
11. Response is Wrong
12. Value of Variable A is 2
13. Give a Hint
14. Student Answers
15. Add 1 to the Value in the Variable
16. Response is Wrong
17. Value of Variable A is 3
18. Write the Correct Answer



The correct feedback can be given in this example because the number of attempts to answer is stored in a variable. Without this number, the decision concerning which feedback should be presented could not be made. Incidentally, note that the value of the variable changed; remember that a variable is called a variable because its value can change (vary).

HOW VALUES OF VARIABLES ARE CHANGED

Values are assigned to (stored in) variables during the run of a program. This is done either by instructions you use in a program or by the PLATO system. Activities 6-B, 6-C, and 6-D discuss this in more detail, describing the variables to which you can assign values, the variables to which the PLATO system can assign values, and the instructions you can use to assign values to variables.

CAI: PROGRAMMING FUNDAMENTALS

This CAI lesson defines the term *variable* and shows you how variables are used to make decisions. The first part of the lesson, "Variables: A Tutorial Review," defines the term. The second part of the lesson, "Flowchart Construction," allows you to use variables to make decisions. As you will see later in this unit, conditional instructions also use variables to make decisions.

Now sign on to a Control Data PLATO terminal using your *student* sign-on. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the title of this lesson, "Programming Fundamentals," press LAB to take the lesson. When you finish the lesson, read the rest of this learning activity.

SUMMARY

The CAI lesson presents the following points:

1. A variable stores things, like a name or a number.
2. Variables have names, like *user*, *a*, or *b*.
3. The things stored in a variable are called *values*. You can assign values to variables (store things in a variable) by using an expression like:

   ```
   a←2
   ```

   The left side of the expression (*a*) is the name of the variable. The ← means *takes as a value*. The *2* is the value assigned to (stored in) the variable.
4. In assigning values, you can use mathematical expressions to represent the value. For example,

   ```
   a←2+2
   ```

   assigns the value *4* to the variable *a*.
5. You can't use mathematical expressions to represent a variable. For example,

   ```
   a+2←2
   ```

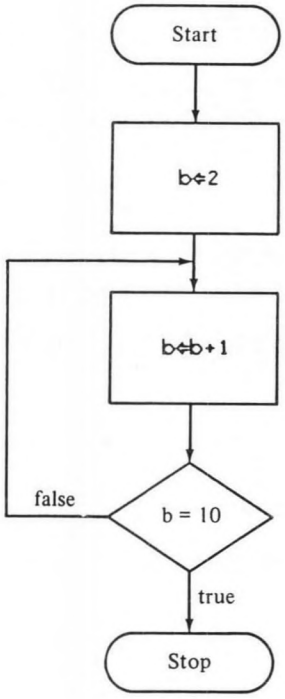
   is illegal.
6. You can use variables as part of a mathematical expression representing a value. The expression

   ```
   a←b+2
   ```

   stores the sum of the value of variable *b* and the number *2* in variable *a*.
7. You can even use the variable on the left side of the expression as part of the representation of the value to be assigned. For example,

   ```
   a←a+2
   ```

   stores the sum of the current value of variable *a* and the numeral *2* in the variable *a*. If the current value of *a* is 2, the expression would assign 4 as the new value of *a*.
8. Variables are used in making decisions. Depending on the current value of a variable, different actions can be performed. In other words, the performance of actions can be *conditional* on the value of a variable. For example, in the following flowchart, the decision to stop or assign a new value to variable *b* is conditional on the value of *b*.



CHECK YOUR UNDERSTANDING

Directions: Mark the following statements *T* for true or *F* for false.

_____ 1. Variables store values: letters, words or numbers.

_____ 2. In the expression a⇦2, *a* is the variable.

_____ 3. a+2⇦2 is a legal expression for assigning values to variables.

_____ 4. a⇦a+2 is a legal expression for assigning values to variables.

_____ 5. In the expression a⇦2, ⇦ means *equals*.

_____ 6. The performance of actions can be conditional on the value of a variable.

Answers

1. T. Values are the contents of variables.
2. T. The left side of the expression is the name of the variable.
3. F. Mathematical expressions cannot be used to represent a variable.
4. T. Variables can be used as part of a mathematical expression representing a value.
5. F. The ⇦ means *takes as a value*.
6. T. Perhaps the most common use of variables in programming is to store information that will be used to make a decision.

6-B. PLATO VARIABLES

Identify characteristics of PLATO variables: programmer-defined variables and system variables.

The Control Data PLATO system allows you to use several different kinds of variables. These variables can be classified into two general categories: programmer-defined variables and system variables.

PROGRAMMER-DEFINED VARIABLES

Programmer-defined means that the programmer can assign values. There are several types of programmer-defined variables. One type is discussed in this activity: student variables. Each student in a course is assigned 15Ø student variables.

In this course, student variables are named by the letter *v* followed by the number of the variable–v1, v2, v3, v4, v5.....v15Ø. The variables assigned to a student are used only by that student and they are used every time the student signs on to the terminal. The value stored in a student variable remains the same until a new value is stored in that variable. These values remain the same even when the student signs off; when the student signs on, the value stored in each variable is the same as the value stored when the student signed off. For example, suppose that a student signs on to a course for the first time and, during the first session, a variable is assigned the value *1*. The student then signs off. When the student signs on the second time, the value stored in the variable is still *1* and will remain *1* until a new value is assigned.

SYSTEM VARIABLES

The PLATO system assigns values to system variables. The programmer *cannot* assign values to system variables. There are several different system variables. In this activity, two are introduced: *ntries* and *where*.

The *ntries* variable records the number of times a student has tried to respond to a question (by entering a response at an arrow). The *ntries* variable can be used to write different messages on each failure to answer a question.

The *where* variable records the coarse-grid location as each character is written. The *where* variable is especially useful in placing text and arrows; it can be the tag for -at- or -arrow-.

The following example illustrates the assignment of values to *ntries* and *where*. Suppose that you are running this program:

```
unit     example
at       2Ø1Ø
write    Who was the first president?
arrow    where+2
specs    bumpshift
answer   washington
wrong    lincoln
at       2Ø3Ø
write    He was the sixteenth president.
endarrow
```

The first thing that happens is that *where* is assigned the value 2Ø1Ø by the -at- instruction. Then this text is written: *Who was the first president*? As each character is written, *where* is incremented by one. The arrow is placed two spaces to the right of the question mark.

You probably noticed that *ntries* doesn't appear in the code. However, because it is a system variable, it still has been defined, and it contains a value: The PLATO system defines and uses all system variables for all programs regardless of whether the variable appears in the code of a particular program.

Now suppose that the student enters *Adams* as a response. The response is judged *no*. Because this is the first response at the arrow, the system sets the value of *ntries* at 1. If the student's next response is *Lincoln*, the response is judged *no*, the message *He was the 16th president*, is written at location 2Ø3Ø, and the value of *ntries* is set at 2. The program continues in this way, adding 1 to the value of *ntries* on each incorrect response, until the student enters the correct response and *ntries* is incremented one last time.

CHECK YOUR UNDERSTANDING

Directions: Select *all* correct answers for the following questions.

_____ 1. Which of the variables listed are programmer-defined variables?

a. v27
b. v13
c. ntries
d. where

_____ 2. Which of the variables listed are system variables?

a. v15Ø
b. where
c. v1
d. ntries

_____ 3. Which of the following variables can be defined by the programmer?

a. v2Ø
b. v16
c. where
d. v2

_____ 4. Which of the following variables records the coarse-grid location as each character of text is written?

a. where
b. ntries
c. v1
d. v18

Answers

1. a,b. Student variables may be defined by the programmer; system variables, like *ntries* and *where*, may not.
2. b,d. v15Ø and v1 are programmer-defined variables.
3. a,b, d. v1,v2,....v15Ø can be defined by the programmer; system variables, like *where*, cannot.
4. a. *Where* records coarse-grid location; *ntries* records number of attempts to answer a question; v1 and v18 are student variables.



6-C. VARIABLES: ASSIGNING VALUES, ASSIGNING NAMES, AND DISPLAYING

Identify the effects of instructions used to assign values to variables, assign names to variables, and display values of variables.

Programmers can assign values to student variables. This learning activity discusses instructions that allow you to do this, as well as instructions that allow you to assign new names to variables and display the values of variables to students.

THE -calc- INSTRUCTION

The -calc- instruction is used to assign numerical values to student variables. The general form of the -calc- instruction is:

```
calc    VARIABLE⇐EXPRESSION
```

The variable is identified by its location, for example, v1, v2, ..., v15Ø. The expression is any mathematical expression defining a numerical value.

For example, the following -calc- instructions assign the values 1Ø.6, 12, and 19 to variables v1, v2, and v5, respectively:

```
calc    v1⇐1Ø.6
calc    v2⇐12
calc    v5⇐19
```

Order of Arithmetic Operations

If the expression defining a value includes arithmetic operations (+, –, ÷, x, and exponentiation), you must be careful about the order in which the operations are applied. For example, in the expression

$$25 + 1Ø \times 2$$

which operation is applied first? If it is the +, the value defined is 7Ø (25 + 1Ø = 35, 35 x 2 = 7Ø). If it is the x, the value defined is 45 (1Ø x 2 = 2Ø, 25 + 2Ø = 45). The value actually defined by this expression is 45, because the x is applied before the +. Table 2 summarizes the order of application of operations.

TABLE 2
Order of Application of Arithmetic Operations

Order	*Operation*	*Example*
First	Exponentiation	2^{3}*
Second	Multiplication (x or *)	2 x 3
Third	Division (÷)	1Ø ÷ 2
Fourth	Addition or Subtraction (+ or –)	2 + 3, or 6 – 2

*Exponents are produced by pressing the SUPER key.

Examples illustrating the order of application of operations are shown below:

Expression	*Value*
$2 + 6^{2}$	38
2×3^{2}	18
1Ø ÷ 2 + 3	8
$24 \div 2^{3}$	3
2 x 3 + 4	1Ø
12 ÷ 2 x 3	2

The rule on order of application of operations is violated only when parentheses are used in the expression. Operations enclosed in parentheses are always performed first. These examples show how parentheses can alter the order of application:

Expression	*Value*
1Ø ÷ (2 + 3)	2
2 x (3 + 4)	14
(12 ÷ 2) x 3	18

Using Variables in Expressions

Variables can be used in expressions defining values as long as the variables have been assigned values. If they have not, you will not know their values and, therefore, you will not know the value of the expression. For example,

```
calc    v1←2
calc    v2←v1+3
```

assigns the value 2 to variable v1 and the value 5 to variable v2. If a value had not been assigned to v1, the value of v1, and hence the value of v2, would be unpredictable.

A variable can also be used in an expression defining the value of the same variable as long as the variable has already been assigned a value. Thus,

```
calc    v1←2
        v1←v1+3
```

assigns the value 5 to variable v1. Note that it is not necessary to repeat the -calc- in the command field if the preceding instruction is also a -calc- instruction.

THE -zero- INSTRUCTION

Suppose that you are using the variable v1 to record the number of correct responses in a quiz. Before presenting the first question, you want to be sure that the variable is set to zero. A non-zero value may have been set earlier in the course. (Remember that values assigned to student variables remain even when the student signs off.) One solution would be to use the -calc- instruction:

```
calc    v1←Ø
```

Now suppose that you are using several variables in a lesson and want to be sure that all are set to zero. You could use a -calc- instruction for each variable, but if there are many variables, this is impractical. A better solution is to use the -zero- instruction.

The general form for -zero- is:

```
zero    VARIABLE,NUMBER OF VARIABLES
```

The variable is the first variable in the sequence. For example, if you are using the variables v1 through v3Ø, v1 is the first variable. The number of variables is the number of variables in the sequence. The instruction

```
zero    v1,3Ø
```

would set variables v1 through v3Ø to zero.

ASSIGNING MEANINGFUL NAMES TO VARIABLES

Although it is possible to write lessons using only the variable names v1, v2..... v15Ø, it is unnecessary. It is also undesirable. When you write programs using similar and meaningless variable names, it is easy to forget what each variable really is. Does v12 record student errors? Or is it v21? This obviously makes it difficult to write programs, and it increases the probability of programming errors.

This is especially true when more than one programmer works on a lesson or when a programmer must work on a lesson after an absence. The problem can be avoided by using the -define- instruction to assign meaningful names to variables.

The general form of -define- is:

```
define  DEFINED NAME = VARIABLE NAME
```

The defined name is the meaningful name assigned to the variable. The variable name is v1, v2.....v15Ø. For example,

```
define  count = v1Ø
```

assigns the name *count* to variable v1Ø. The variable can be referred to as *count* or v1Ø throughout the rest of the program; however, to minimize probability of programming error, *count* is preferred.

There are rules for choosing defined names: The name must be seven characters or less; it must not begin with a number; and it must not contain an arithmetic operation symbol. For example, count and count2 are legal names, but counttwo, 2count, and count+ are not.

The -define- instruction must precede any reference to the defined name. A good practice is to place all -define- instructions before the first -unit- instruction. This ensures their execution (and makes them easy to locate). The instructions placed before the first -unit- instruction are called the *initial entry unit* (IEU). These instructions are always executed first when any lesson is entered, no matter which unit is entered first. (In *PLATO Author Language: Part II*, you'll learn how to begin a lesson at places other than the first physical unit.) No -unit- instruction should precede instructions, such as -define-, that are in the IEU.

Defined names can be used in -calc- instructions in the same way as v1, v2, or v3. For example,

```
define  count = v1
unit    example
calc    count←Ø
```

sets the value of the variable *count* (v1) to Ø. Note that the -define- instruction precedes the first -unit- instruction: It is in the IEU.

Defined names can also be used in -zero- instructions. For example,

```
define  count = v1
unit    example
zero    count,1Ø
```

sets the value of v1 through v1Ø (the 1Ø variables starting with v1) to zero.

It is possible to define more than one name in the same line by separating the defined name and variable name pairs by a comma. For example:

```
define  count = v1,right = v2,wrong = v3
```

Be careful with this use of -define-, however, because it makes changing the instruction more difficult; if you want to change one defined name and variable name pair, you must retype all of them. The alternative is to name each variable on a separate line:

```
define  count = v1
        right = v2
        wrong = v3
```

It is not necessary to repeat the -define- command if subsequent instructions are also -define- instructions.

THE -show- INSTRUCTION: DISPLAYING VARIABLES

As mentioned earlier, variables can be used to make decisions. You will see how this is done in the discussion of conditional instructions later in the unit. There is another important use of variables that should be mentioned, however: You can use variables to store information you want to display to the student. For example, you may want to store the number of correct responses a student has made on a quiz in order to write a display showing the number of correct responses at the end of the quiz.

The -show- instruction is used to display the value of a variable to a student. The general form for -show- is:

```
show    VARIABLE
```

The variable stores the value to be shown. For example, if variable v5 has the value 15, this code

```
at      2Ø1Ø
write   YOU MADEØ
show    v5
write   ØCORRECT RESPONSES.
```

will result in the following display when executed:

```
YOU MADE 15 CORRECT RESPONSES.
```

Note that spaces must be added to the -write- instructions before and after the -show- instruction. (This is done by pressing the space bar on the keyset. The space symbol indicates where the space bar is pressed.) -Show- doesn't add spaces.

If the spaces were not added, the message would look like this:

```
YOU MADE15CORRECT RESPONSES.
```

You can also show variables with defined names. For example, if *count* has been defined as v5, this code

```
at      2Ø1Ø
write   YOU MADE□
show    count
write   □CORRECT RESPONSES.
```

is equivalent and preferable to the code using the variable v5.

It is possible to reduce the number of lines of code needed to show a variable by *embedding* the -show- instruction in the -write- instruction surrounding it. The lines

```
at      2Ø1Ø
write   YOU MADE ◊s,count◊ CORRECT RESPONSES.
```

are equivalent to the four lines immediately above. The line

```
show    count
```

has been embedded in a single -write- instruction by replacing the command *show* with the letter *s*, separating the command from the tag (*count*) with a comma, and enclosing the command and tag in ◊ ◊ symbols. The ◊ symbol is obtained by pressing the shifted □ key, followed by the number Ø. The ◊ is obtained by pressing the shifted □ key followed by the numeral 1.

CHECK YOUR UNDERSTANDING

Section 1

Directions: Select *all* the correct answers for the following questions.

_____ 1. What is the value of v1?

```
calc    v1←18÷3×2
```

a. 12
b. 3
c. 4
d. 6

_____ 2. What is the value of v1?

```
calc    v1←6
        v1←v1×2+4÷2
```

a. 8
b. 18
c. 14
d. 6

_____ 3. Which variables are set to zero?

```
zero    v4,1Ø
```

a. v4 through v13
b. v4 through v14
c. v4 and v1Ø
d. v4 and v14

_____ 4. What is the value of v6?

```
define  error = v6
unit    example
calc    error←Ø
```

a. the value of define
b. the value of error
c. the value of example
d. Ø

_____ 5. What is written?

```
define   error = v6
unit     example
calc     error←Ø
at       2Ø1Ø
write    YOU HAVE ‹s,error› ERRORS.
```

a. YOU HAVE Ø ERRORS.
b. YOU HAVEØERRORS.
c. YOU HAVE s,Ø ERRORS.
d. YOU HAVE ZERO ERRORS.

Answers

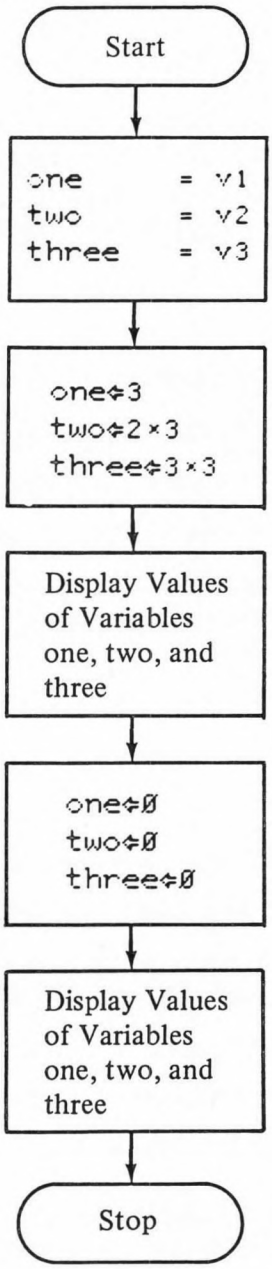
1. b. Multiplication is performed before division.
2. c. Multiplication is performed first, then division, then addition.
3. a. The 1Ø variables following v4, *including v4*, are set to zero.
4. b,d. *Errǫr* is just another name for v6; they are the same variable.
5. a. The value of variable *error* (Ø) is shown; spaces are included before and after the embedded show.

Section 2

Directions: Write and execute a program that:

1. Assigns the names one, two, and three to variables v1, v2, and v3, respectively
2. Assigns a non-zero value to variable *one*, assigns two times this value to variable *two*, and three times this value to variable *thr̄ē*
3. Displays the values of the three variables
4. Sets the value of each of the three variables to zero
5. Displays the new values of the three variables

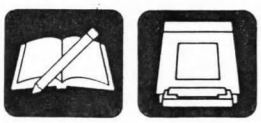
Use -calc-, -zero-, -define-, and -show-. Structure your program according to the following flowchart. One possible solution is given in the answer key.


Start
one = v1
two = v2
three = v3
one⇦3
two⇦2×3
three⇦3×3
Display Values of Variables one, two, and three
one⇦Ø
two⇦Ø
three⇦Ø
Display Values of Variables one, two, and three
Stop

Answer Key

```
define  one      = v1
        two      = v2
        three    = v3
unit    example
calc    one←3
        two←2×one
        three←3×one
at      405
write   The value of:
        one is ⟨s,one⟩
        two is ⟨s,two⟩
        three is ⟨s,three⟩
zero    one,3
at      1005
write   The value of:
        one is ⟨s,one⟩
        two is ⟨s,two⟩
        three is ⟨s,three⟩
```

6-D. RANDOM NUMBER GENERATION



Identify the effects of random number generation instructions.

Programmers can set the values of student variables using the -calc- instruction or, if the value is to be zero, the -zero- instruction. If the programmer doesn't need to specify the values of variables exactly, random number generation instructions can also be used. For example, a programmer writing an arithmetic drill may want to use any numbers less than 1ØØ. Random number generation instructions can generate such numbers and assign them to variables specified by the programmer. This activity discusses two random number generation instructions: -randu- and -randp-.

THE -randu- INSTRUCTION

It is helpful to think of random number generation as selecting a number at random from a defined pool of numbers. The pool of numbers contains all numbers between a lower limit and an upper limit. For example, the pool with a lower limit of 1 and an upper limit of 9 contains all the numbers from 1 up to and including 9.

The general form of -randu- is

```
randu    VARIABLE,UPPER LIMIT
```

The variable is the variable in which the generated number is to be stored. The upper limit specifies the upper limit of the pool of numbers from which the generated number is to be selected. The lower limit is always 1. For example,

```
randu    v1,9
```

stores a number from 1 to 9 in variable v1.

The upper limit can be expressed as another different variable. For example:

```
calc     v2=9
randu    v1,v2
```

or

```
define   limit    = v2
unit     example
calc     limit=9
randu    v1,limit
```

Every number in the pool has an equal chance of being selected each time the -randu- instruction is executed. For example, if 2 is selected on the first execution, it is just as likely as all the other numbers in the pool to be selected on the second execution. This is called *sampling with replacement*: a selected (sampled) number is replaced in the pool for the next selection.

THE -randp- INSTRUCTION

Sometimes sampling with replacement is inappropriate. For example, suppose that you don't want to repeat numbers in an arithmetic drill; each number is to occur once and only once. The -randp- instruction samples without replacement. When a number is selected, it is removed from the pool and cannot be selected again.

In order to use -randp-, you must also use -setperm-. The -setperm- instruction creates the pool of the numbers from which -randp- selects. The general form of the -setperm- is:

```
setperm UPPER LIMIT
```

The upper limit is the upper limit of the pool. The lower limit is always 1. For example,

```
setperm 9
```

creates a pool containing the numbers 1 through 9. As with -randu-, the upper limit can be expressed as a variable. For example,

```
calc    v2←9
setperm v2
```

The general form of -randp- is:

```
randp   VARIABLE
```

The variable is the variable in which the generated number is to be stored. For example,

```
setperm 9
randp   v1
```

stores a number from 1 to 9 in variable v1. Repeated executions of -randp- will never duplicate selected numbers. When all numbers in the pool have been selected, -randp- stores a ∅. The pool can be renewed by repeating an execution of -setperm-.

A common error is repeating the execution of -setperm-, and therefore renewing the pool, before all of the numbers desired have been selected. Repeating the execution of -setperm- too early destroys sampling without replacement; numbers already selected can be selected again. For example,

```
setperm 9
randp   v1
setperm 9
randp   v1
```

stores a number from 1 to 9 in variable v1 on the first execution of -randp-. Because -setperm- is executed again before the second -randp-, the pool is renewed and the second -randp- will store a number from 1 to 9 in variable v1. The sampling without replacement has been destroyed.

CHECK YOUR UNDERSTANDING

Section 1

Directions: Select *all* the correct answers for the following questions.

_____ 1. Which instructions could set the value of v1 at 6?

a.
```
randu   v1,4
```

b.
```
setperm 10
randp   v1
```

c.
```
randu   v1,8
```

d.
```
setperm 3
randp   v1
```

_____ 2. Which assignments of values to variables could be produced by this sequence of instructions?

```
randu   v1,4
randu   v2,4
randu   v3,4
randu   v4,4
```

a. v1⇐2, v2⇐4, v3⇐2, v4⇐1

b. v1⇐1, v2⇐2, v3⇐3, v4⇐4

c. v1⇐2, v2⇐5, v3⇐2, v4⇐1

d. v1⇐4, v2⇐3, v3⇐2, v4⇐1

_____ 3. Which assignments of values to variables could be produced by this sequence of instructions?

```
setperm 4
randp   v1
randp   v2
randp   v3
randp   v4
```

a. v1⇐2, v2⇐4, v3⇐2, v4⇐1

b. v1⇐1, v2⇐3, v3⇐4, v4⇐2

c. v1⇐5, v2⇐3, v3⇐1, v4⇐Ø

d. v1⇐4, v2⇐3, v3⇐1, v4⇐2

Answers

1. b,c. The upper limit in the tag of -randu- and -randp- specifies the highest number that can be selected.

2. a,b, d. -randu- can select numbers which have already been selected, but the numbers must not be greater than the specified upper limit.

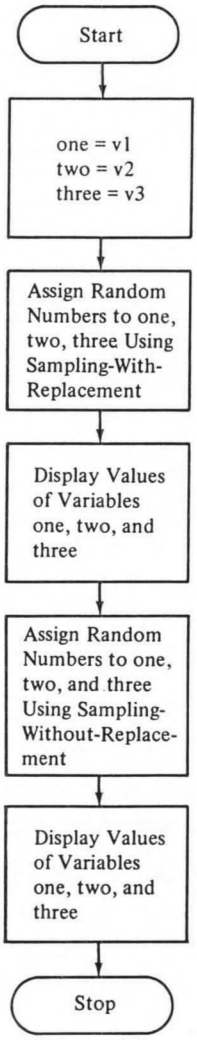
3. b,d. -randp- will not select numbers which have already been selected. The numbers cannot be higher than the specified upper limit.

Section 2

Directions: Write and execute a program that:

1. Assigns the names *one*, *two*, and *three* to variables v1, v2, and v3, respectively
2. Assigns values to variables *one*, *two*, and *three* using a sampling-with-replacement random-number generator
3. Displays the values of the three variables
4. Assigns values to variables *one*, *two*, and *three* using a sampling-without-replacement random-number generator
5. Displays the new values of the three variables

Use -randu-, -setperm-, -randp-, and -show-. Structure your program according to the following flowchart. One possible solution is given in the answer key.



Answer Key

```
define  one     = v1
        two     = v2
        three   = v3
unit    example
randu   one,3
randu   two,3
randu   three,3
at      405
write   The value of:
        one is (s,one)
        two is (s,two)
        three is (s,three)
setperm 3
randp   one
randp   two
randp   three
at      1005
write   The value of:
        one is (s,one)
        two is (s,two)
        three is (s,three)
```



6-E. CONDITIONAL INSTRUCTIONS: WRITING, JUDGING, AND CALCULATING

Identify the effects of -writec-, -answerc-, -wrongc-, and -calcs-.

Perhaps the most important reason for using variables in programming is that they allow your programs to make decisions on what to do next. These decisions are dependent (or conditional) on the values of variables. Your programs need not be restricted to a linear flow of actions (*First do this, then do that*). With decision-making capability, your programs can perform one action *or* another (*If this is true, do this; if not, do that*). This allows you to write more flexible and powerful programs.

Many of the instructions you have already seen can be written in a conditional form. For some, like -write-, -answer-, -wrong-, and -calc-, there are parallel conditional instructions that have effects similar to the unconditional instructions, but that are written in a conditional form. This activity begins with a discussion of the general form for conditional instructions. It then presents a discussion of the conditional parallels of -write-, -answer-, -wrong-, and -calc-.

THE CONDITIONAL FORM

In the conditional form, the effect of an instruction depends on the value of a specified variable at the time of execution. The general form for all conditional instructions is:

```
COMMAND EXPRESSION,ALTERNATIVE 1,ALTERNATIVE 2, ...
```

The tag contains an expression followed by several alternatives. The expression and each alternative are separated by *delimiters*. There are several possible delimiters. Commas are used for now, and other delimiters are discussed later in this activity. The delimiter separating the expression from the first alternative must be used to separate the alternatives; the PLATO system assumes that the first delimiter in a conditional instruction is the only delimiter used in the instruction.

The expression may be a variable or an arithmetic expression containing variables and arithmetic operators. The value of the expression determines which alternative is to be performed. The relation of expression value to alternative is summarized in table 3. Any negative value causes the first alternative to be performed, $\emptyset$ causes the second alternative to be performed, and any positive value (n) causes

alternative n + 2 to be performed. For example, if the value is 1, alternative 3 (1 + 2) is performed. If there is no alternative given for a value, the last alternative is performed. For example, if the value of the expression is 3 (calling for the performance of alternative 5) and only four alternatives are given, then alternative 4 is performed.

TABLE 3
Relation of Expression Value and Alternative

Expression Value	Negative	0	1	2	3	...	n
Alternative	1	2	3	4	5	...	n + 2

THE -writec- INSTRUCTION

The -writec- instruction is the conditional parallel of the unconditional -write- instruction. Alternatives are alternative messages to be written:

```
writec  EXPRESSION,MESSAGE 1,MESSAGE 2, ...
```

If the expression is negative, message 1 is written, if it is ∅, message 2 is written, and so on.

It will usually be impossible to fit all of the alternative messages into a single line of code. If all messages cannot be written on a single line, each message should begin on a separate line. For example:

```
writec  EXPRESSION,
        MESSAGE 1,
        MESSAGE 2,
          .
          .
          .
```

This will make it easier for you to format your messages since it will be easier to control where each line of a message will end.

You can use multiple lines for a single message just as in the unconditional -write- instruction. Remember that if you go to a new line by pressing NEXT and then TAB, a new line will be written when the instruction is executed.

As it has been defined, the conditional form uses commas as delimiters setting alternative actions apart from one another. With -writec-, this may produce problems because the alternative actions are messages that may themselves contain commas. Other possible delimiters for -writec- are the semicolon or the end of a

line. Neither is satisfactory. The semicolon, like the comma, may be part of a message; the end of a line as a delimiter requires that messages be no longer than one line.

The use of the universal delimiter ✱ in all -writec- instructions is recommended. The delimiter will not be part of a message, and it allows the use of multiple-line messages. You produce the ✱ by first pressing the ACCESS key (shifted-□) and then pressing the comma key. An example of the use of -writec- is shown below:

```
unit     example
at       2Ø1Ø
write    Who was the first president?
arrow    where+2
specs    bumpshift
answer   washington
write    That's right!
no
at       2Ø2Ø
writec   ntries-2✱No, try again.✱
         His picture is on the dollar bill.✱
         Try Washington.
endarrow
```

The expression uses the system variable *ntries*, the number of responses the student has made to a question. After the first response, *ntries* = 1, so the expression (*ntries* – *2*) = –1. This causes the printing of *No, try again*. After the second response, *ntries* = 2, the expression = Ø, and *His picture is on the dollar bill* is printed. After the third response, and every response thereafter, the message *Try Washington* is printed.

Note that *ntries* – *2* was used as the expression in order to produce values of –1, Ø, and 1 as required by -writec-. The actual values of *ntries*, 1, 2, and 3, would cause the writing of only the third message, *Try Washington*. (Remember that if there is no alternative given for a value, the last alternative is performed). Since many of the variables used in conditional instruction expressions have values no lower than 1 or Ø, this is a common problem. Subtracting to produce a lowest value of –1 is one solution. Another solution is to use null (empty) alternatives in the tag as space fillers. Two delimiters with nothing between them define a null alternative. This is achieved by typing one delimiter immediately after another one, with nothing between. Since there is no alternative to perform, nothing happens. The -writec- in the example could have been written as follows:

```
writec   ntries✱✱✱No, try again.✱
         His picture is on the dollar bill.✱
         Try Washington.
```

If ntries = 1, 2, or 3, the first, second, or third message is written.

Null alternatives can be used anywhere in the list of alternatives. For example, if you wanted messages written only when *ntries* = 1 and 3, you could use:

```
writec  ntries***No, try again.**
        Try Washington.
```

CHECK YOUR UNDERSTANDING

Section 1

Directions: Questions 1 through 3 refer to the following instruction. Choose the best answers for each question.

```
writec  count**
        MESSAGE ONE*
        MESSAGE TWO*
        MESSAGE THREE
```

_____ 1. What message would be written if the value of *count* is –1?

a. Nothing
b. MESSAGE ONE
c. MESSAGE TWO
d. MESSAGE THREE

_____ 2. What message would be written if the value of *count* is 2?

a. Nothing
b. MESSAGE ONE
c. MESSAGE TWO
d. MESSAGE THREE

_____ 3. What message would be written if the value of *count* is 4?

a. Nothing
b. MESSAGE ONE
c. MESSAGE TWO
d. MESSAGE THREE

Directions: Questions 4 and 5 refer to the following instructions. Choose the best answer for each question.

```
writec  count,Pearls of wisdom,
        winners never quit, quitters never win,
        when the going gets tough, the tough get going
```

_____ 4. What message would be written if the value of *count* is 1?

a. Pearls of wisdom
b. when the going gets tough, the tough get going
c. quitters never win
d. winners never quit

_____ 5. What message would be written if the value of *count* is 2?

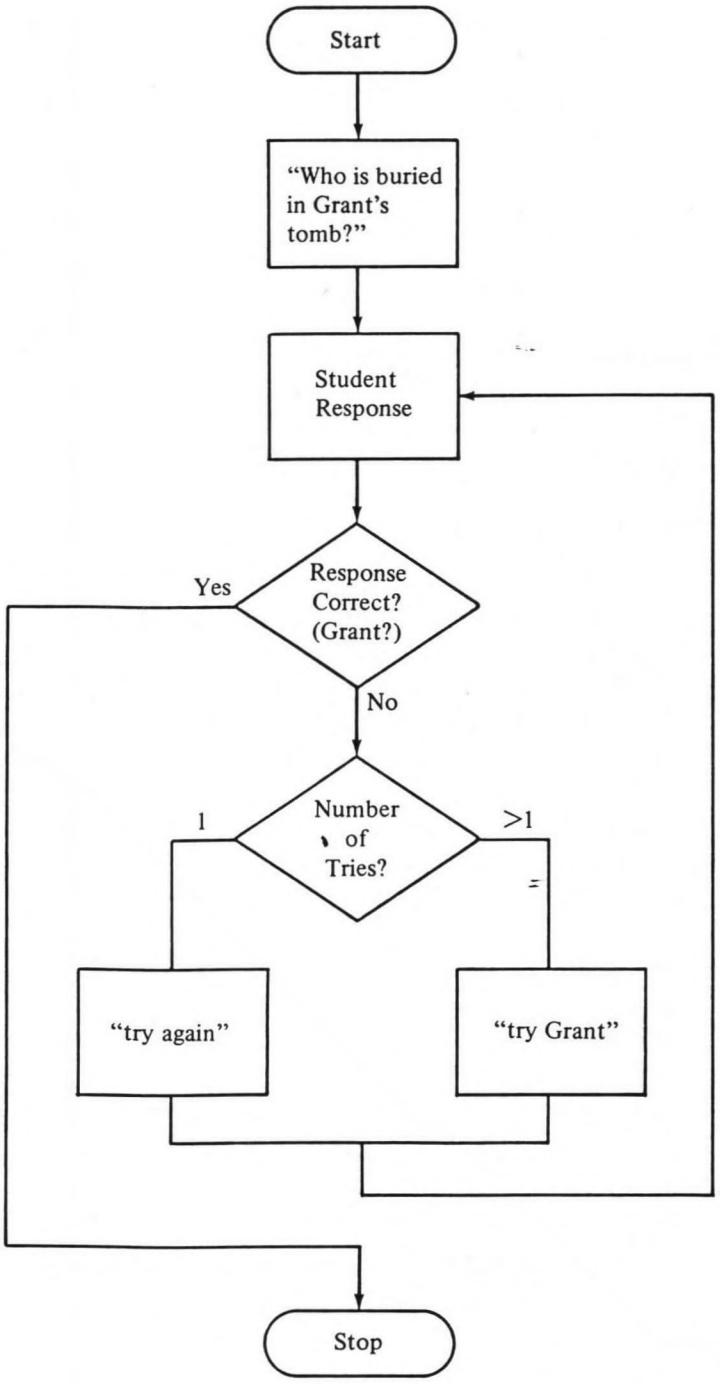
a. when the going gets tough, the tough get going
b. winners never quit, quitters never win
c. when the going gets tough
d. winners never quit

Answers

1. a. The alternative when *count* is -1 is null (empty).
2. d. When *count* is 2, the fourth alternative is performed.
3. d. When there is no alternative given, the last alternative is performed.
4. c. Since the delimiter is a comma, five alternatives are given. When *count* is 1, the third alternative is performed.
5. c. The delimiter is a comma. When *count* is 2, the fourth alternative is performed.

Section 2

Directions: Write and execute a program that asks a question, writes *try again* after the first incorrect response, and writes the correct answer after the second incorrect response. Use -writec- and *ntries*. Structure your program according to the following flowchart. One possible solution is given in the answer key.



Answer Key

```
unit     example
at       2Ø1Ø
write    Who is buried in Grant's tomb?
arrow    where+2
specs    bumpshift
answer   grant
no
at       231Ø
writec   ntries-2♦try again♦try Grant
endarrow
```

THE -answerc- AND -wrongc- INSTRUCTIONS

The -answerc- and -wrongc- instructions are the conditional parallels of the unconditional -answer- and -wrong- instructions. Alternatives are alternative answers.

```
answerc EXPRESSION♦ANSWER 1♦ANSWER 2♦ ...
wrongc  EXPRESSION♦ANSWER 1♦ANSWER 2♦ ...
```

Each alternative answer can consist of a single word or phrase or several synonyms. Synonyms are separated by a comma. Because of this, commas cannot be used as delimiters setting apart alternative answers. Alternative answers are separated by a semicolon or the universal delimiter ♦. As with -answer- and -wrong-, ignorable words are enclosed by < > symbols.

An example of the use of -answerc- and -wrongc- is shown below. Note that this example also uses a -define- instruction to name the variable in the expression. The -randu- and -writec- instructions are also used.

```
define  problem = v1
unit    example
randu   problem,2
at      2Ø1Ø
writec  problem-2♦Who was the first president?♦
        Who was the sixteenth president?
arrow   221Ø
specs   bumpshift
answerc problem-2♦<george> washington♦
        <abraham,abe> lincoln
write   That's right!
wrongc  problem-2♦<abraham,abe> lincoln♦
        <george> washington
writec  problem-2♦No, he was the sixteenth president.♦
        No, he was the first president.
endarrow
```

The -randu- instruction generates a 1 or 2 which is stored in the variable *problem*. Depending on the value of *problem*, one of two questions are asked at the first -writec-. The *problem* variable is also used in the -answerc- and -wrongc- instructions and in the second -writec-. If the value of *problem* is 1, the program asks *Who was the first president*? If the response is *Washington*, there is a match at the -answerc-, the response is judged *ok*, and the program writes *That's right*! If the response is *Lincoln*, there is a match at the -wrongc-, the response is judged *no*, and the program writes *No, he was the sixteenth president*. As with the -writec- example discussed earlier, 2 is subtracted from the variable (*problem*) in order to obtain values of –1 and ∅. Null alternatives could also be used.

CHECK YOUR UNDERSTANDING

Section 1

Directions: Questions 1 and 2 refer to the following instruction. Choose the correct answers for each question.

```
answerc count-2♦New York, New York City♦
        Los Angeles, LA
```

_____ 1. What answer would produce a match if the value of *count* is 1?

a. New York
b. New York City
c. LA
d. Los Angeles

_____ 2. Identify the instructions that are equivalent to:

```
answerc count-2♦New York, New York City♦Los Angeles, LA
```

a.
```
answerc count♦♦New York,= New York City♦
        Los Angeles, LA
```

b.
```
answerc count♦♦♦New York, New York City♦
        Los Angeles, LA
```

c.
```
answerc count-2;New York, New York City;
        Los Angeles, LA
```

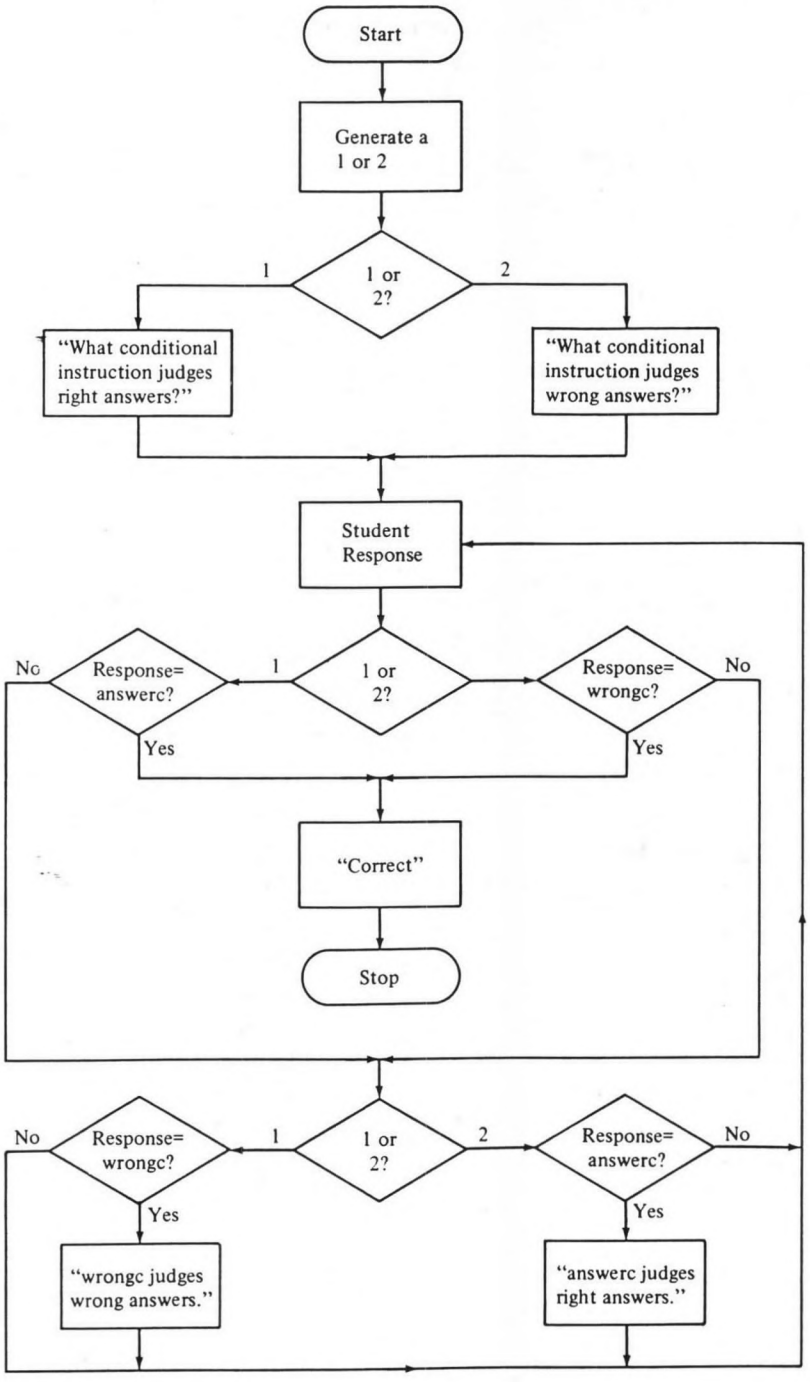
d.
```
answerc count;;New York, New York City;
        Los Angeles, LA
```

Answers

1. a,b 2. b, c

Section 2

Directions: Write and execute a program that asks one of two possible questions and gives appropriate feedback for anticipated right and wrong answers. Use -writec-, -answerc-, and -wrongc-. Structure your program according to the following flowchart. One possible solution is given in the answer key.



Answer Key

```
define  problem = v1
unit    example
randu   problem,2
at      2Ø1Ø
writec  problem-2*
        What conditional instruction judges right
        answers?*
        What conditional instruction judges wrong
        answers?
arrow   where+2
answerc problem-2*answerc*wrongc
at      221Ø
write   Correct
wrongc  problem-2*wrongc*answerc
writec  problem-2*
        wrongc judges wrong answers*
        answerc judges right answers
endarrow
```

THE -calcs- INSTRUCTION

The -calcs- instruction is a conditional parallel of the unconditional -calc- instruction. It is used when you want to assign one of several values to a variable, conditional on the value of another variable. For example, suppose that you want to increment the number of wrong answers on a student's first failure to correctly answer a question. You could use a -calc- to increment the value of the variable storing the number of wrong answers after every *no* judgment. Since there may be several *no* judgments (repeated failures to answer correctly) for each question, however, you would be incrementing the variable after *every* wrong answer and there may be more wrong answers than there are questions. The -calcs- instruction allows you to increment the variable only after the first failure to answer each question correctly; a value can be assigned to the variable storing wrong answers conditional on the value of *ntries*. You will use -calcs- in this way in the exercise following this section. The general form for -calcs- is:

```
calcs   EXPRESSION,VARIABLE*VALUE1,VALUE2,...
```

Alternatives are alternative values to be assigned to a variable. Commas are used as delimiters.

If the expression is negative, value 1 is assigned to the variable; if it is Ø, value 2 is assigned; and so on. For example, execution of the instructions

```
define  count   = v1
        test    = v2
  .
  .
unit    example
calc    count≑Ø
calcs   test,count≑count-1,count+1
```

would set the value of the variable count at –1 if the variable *test* is negative and at 1 if *test* is zero or positive.

Alternatives can be null (empty). For example, if you replaced the -calcs- instruction in the preceding example with

```
calcs   test,count≑,count-1,count+1
```

the value of *count* would be set at –1 if *test* is zero and at 1 if *test* is positive. If *test* is negative, the value of *count* would not be changed.

The -calcs- instruction may be confusing because two variables are being used in the tag. One is the expression, which may be a variable (*test*) or an expression that contains a variable (*test–2*, for example); the other is the variable in which a value is to be stored. Remember that the *first* variable in the tag of -calcs- is the expression and the second is the variable in which a value is to be stored. The value stored in the first variable (the expression) indicates which value is to be stored in the second variable.

CHECK YOUR UNDERSTANDING

Section 1

Directions: Question 1 refers to the following instruction. Choose the best answer.

```
define  test    = v1
        count   = v2
calc    test÷1
  .
  .
  .
calcs   count,test÷test+1,test+2
```

_____ 1. What is the value assigned to the variable *test* if the value assigned to *count* is zero?

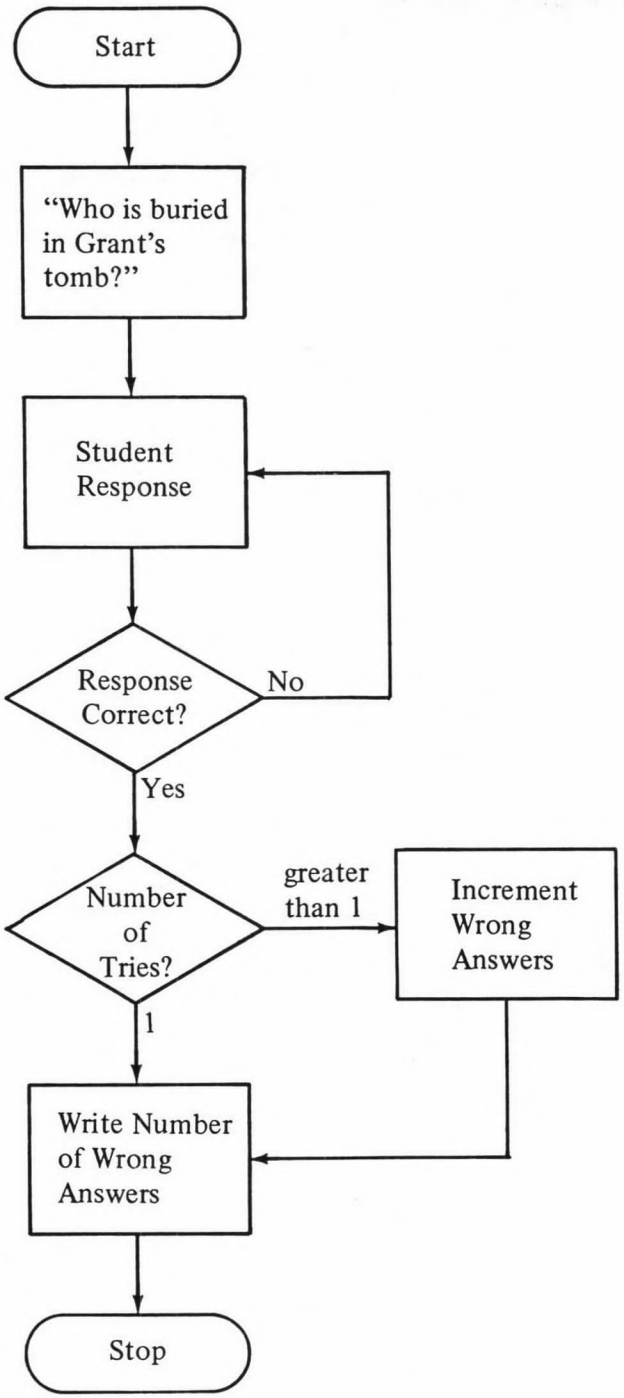
a. 1
b. 2
c. 3
d. nothing

Answer

1. c. If the expression is ∅, the second alternative is performed and determines the value of variable *test* (test + 2 = 3).

Section 2

Directions: Write and execute a program that asks a question, judges the student's response, and adds 1 to the value of a variable recording number of wrong answers when the student's first response is incorrect. The program should write the new value of the variable that records the number of wrong answers so that you can determine whether the program is working correctly. Use -calcs- and -writec-. Structure your program according to the following flowchart. One possible solution is given in the answer key.


Start
"Who is buried in Grant's tomb?"
Student Response
Response Correct?
No
Yes
Number of Tries?
greater than 1
Increment Wrong Answers
1
Write Number of Wrong Answers
Stop

Answer Key

```
define  wrong    = v1
calc    wrong←0
unit    example
at      2010
write   Who is buried in Grant's  tomb?
arrow   where+2
specs   bumpshift
answer  grant
endarrow
calcs   ntries-2,wrong←,wrong+1
at      2210
write   number wrong = (s,wrong)
```

Note the comma used in the -calcs- instruction. It allows for the null alternative; *wrong* will not be incremented if ntries = 1.



6-F. CONDITIONAL INSTRUCTIONS: BRANCHING BETWEEN UNITS

Identify the effects of conditional forms of -next-, -back-, -stop-, -jump-, -do-, -join-, and -goto-.

Unit 4 discusses the unconditional forms of instructions that allow you to branch between units: -next-, -back-, -stop-, -jump-, -do-, and -join-. This activity discusses the conditional forms of these instructions and introduces a new conditional instruction that allows branching between units: -goto-.

THE -next-, -back-, AND -stop- INSTRUCTIONS

The unconditional forms of -next-, -next1-, -back-, -back1-, and -stop- are discussed in activity 4-A. These instructions can also be used in a conditional form:

```
next     EXPRESSION,UNIT1,UNIT2,UNIT3, ...
next1    EXPRESSION,UNIT1,UNIT2,UNIT3, ...
back     EXPRESSION,UNIT1,UNIT2,UNIT3, ...
back1    EXPRESSION,UNIT1,UNIT2,UNIT3, ...
stop     EXPRESSION,UNIT1,UNIT2,UNIT3, ...
```

As with all other conditional instructions, the value of the expression indicates the alternative performed. The alternatives are units. If the value of the expression is negative, the appropriate key press will cause a branch to the first unit in the list; if it is Ø, the key press will cause a branch to the second unit; and so on.

The following code is an example of the use of the conditional -next- (-next1-, -back-, -back1-, and -stop- would have the same effect):

```
define   index    = v1
unit     main
   .
   .
   .
next     index,one,two,three
   .
   .
   .
```

```
unit    one
  .
  .
  .
unit    two
  .
  .
  .
unit    three
```

If *index* has a negative value when the -next- instruction is executed, the program will branch to *unit one* when NEXT is pressed at the end of *unit main*. If *index* = ∅, the branch will be to *unit two*. If *index* = 1, the branch will be to *unit three*.

Null alternatives *cannot* be used in these instructions. Instead, the letter *x* can be inserted as a "space filler" if you don't want to branch to another unit for some values of the expression. For example, the instruction

```
back    index,one,x,two
```

would, given a BACK key press, branch to *unit one* if the variable *index* were negative when the -back- instruction was executed and to *unit two* if *index* were positive. If the value of *index* were zero, there would be no branch; the *x* selected by the conditional would cause the instruction to have no effect. The letter *x* is always used for this purpose; it can never be used as the name of a unit.

The branch is based on the value of the expression when the instruction activating the key is executed, not when the key is pressed. Because of this you should be sure to assign values to variables in the expression *before* the instruction activating the key is executed. *Place all -calc- instructions assigning values to variables so that they are executed before the function-key instruction.*

CHECK YOUR UNDERSTANDING

Section 1

Directions: Questions 1 and 2 refer to the following instructions. Choose the best answer for each question.

```
unit    begin
  .
  .
next    count,add,x,divide
  .
  .
```

_____ 1. What happens at the end of *unit begin* if the value of *count* is –1 when the -next- instruction is executed and NEXT is pressed?

a. A branch to *unit add*
b. A branch to *unit x*
c. A branch to *unit divide*
d. The *x* selected by the conditional will cause the -next- instruction to have no effect; when NEXT is pressed, the unit following *unit begin* will be executed.

_____ 2. What happens at the end of *unit begin* if the value of *count* is ∅ when the -next- instruction is executed and NEXT is pressed?

a. A branch to *unit add*
b. A branch to *unit x*
c. A branch to *unit divide*
d. The *x* selected by the conditional will cause the -next- instruction to have no effect; when NEXT is pressed, the unit following *unit begin* will be executed.

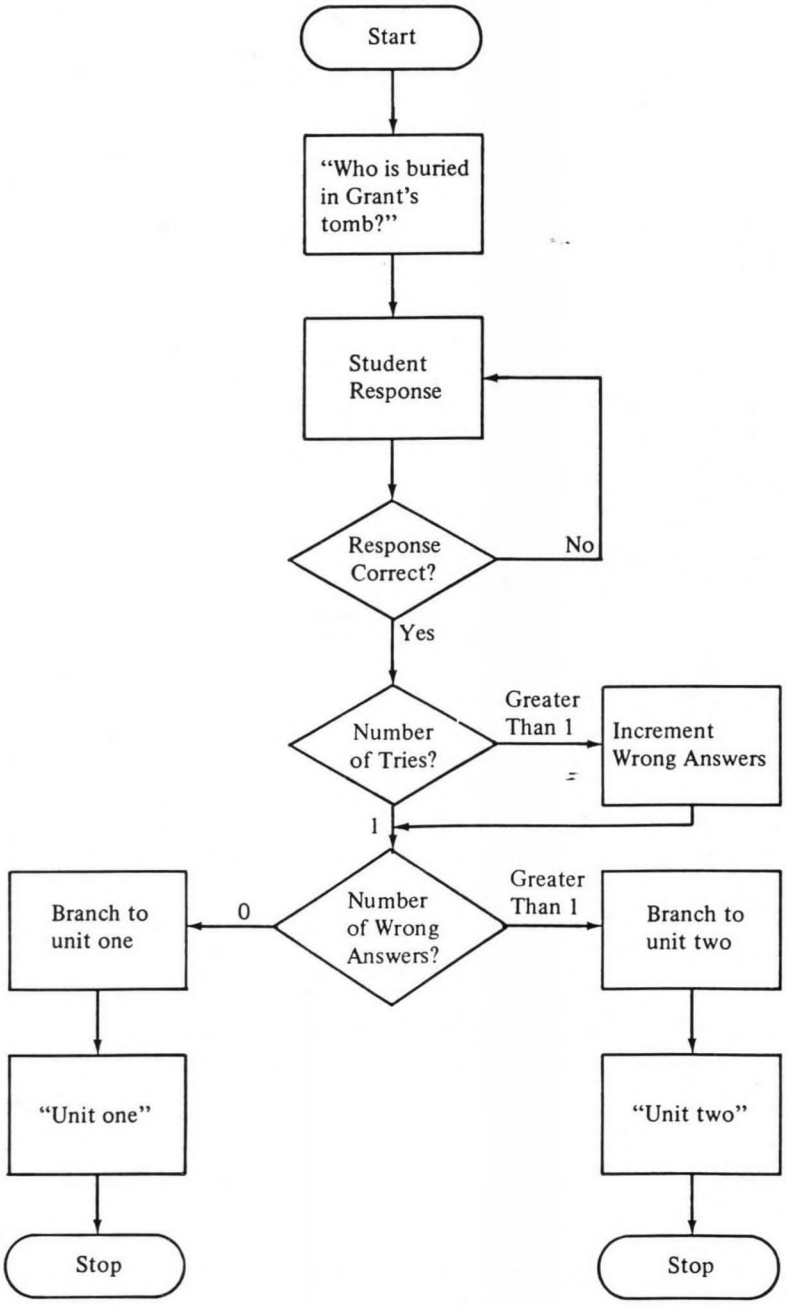
Answers

1. a. If the value of the expression is –1, the branch will be to the first unit listed in the tag.

2. d. If the conditional selects the *x*, the instruction has no effect and the next physical unit is executed just as it would be if there were no -next- instruction.

Section 2

Directions: Write and execute a program that branches from one unit to one of two other units when the NEXT key is pressed. The unit to which the branch is made depends on whether the student's first response to a question is correct. Each unit should write its name so that you can determine whether the program is working correctly. Use the conditional form of -next- and -calcs-. Structure your program according to the following flowchart. One possible solution is given in the answer key. Save your solution for use in the next exercise.



Answer Key

```
define  wrong   = v1
calc    wrong←Ø
unit    example
at      2Ø1Ø
write   Who is buried in Grant's tomb?
arrow   where+2
specs   bumpshift
answer  grant
endarrow
calcs   ntries-2,wrong←,wrong+1
next    wrong-1,one,two
unit    one
at      221Ø
write   unit one
unit    two
at      221Ø
write   unit two
```

THE -jump-, -do-, AND -join- INSTRUCTIONS

The unconditional forms of -jump-, -do-, and -join- are discussed in activity 4-C. The conditional form of these instructions is similar to that of -next-, -next1-, -back-, -back1-, and -stop-:

```
jump    EXPRESSION,UNIT1,UNIT2, ...
do      EXPRESSION,UNIT1,UNIT2, ...
join    EXPRESSION,UNIT1,UNIT2, ...
```

These instructions have the same effects as their unconditional forms except that the value of the expression determines the unit to which the branch is made. As with -next-, -next1-, -back-, -back1-, and -stop-, the letter *x* can be used as a space filler if there is a value of the expression that should cause the instruction to be ignored. Null alternatives cannot be used.

THE -goto- INSTRUCTION

The -goto- instruction is introduced for the first time in this unit because it is almost always used in its conditional form. The conditional form of -goto- is:

```
goto    EXPRESSION,UNIT1,UNIT2, ...
```

Its effect is similar to that of -jump-, -do-, and -join-; a negative value in the expression causes a branch to the first unit, Ø causes a branch to the second unit, 1 causes a branch to the third unit, and so on.

Like -do- and -jump-, and unlike -join-, -goto- is executed only in the regular state. Like -do- and -join-, and unlike -jump-, the unit branched to is an attached or auxiliary unit and there is no full-screen erase. Unlike -do- and -join-, however, there is no return to the instructions in the main unit that follow the branch; the instructions following the -goto- are not executed if there is a branch. The return is to the unit physically following the main unit unless a -next- instruction is included in the auxiliary unit. If no branch is desired for some values of the expression, the letter *x* can be used as a space filler. Null alternatives cannot be used.

The following instructions illustrate the use of -goto- in branching to new units:

```
unit    test1
  .
  .
goto    correct-1,remed,remed,x
at      2Ø1Ø
write   Good work!
  .
  .
unit    test2
  .
  .
unit    remed
  .
  .
```

Assume that the student has received a quiz. The variable *correct* records the number of correct responses. The -goto- instruction branches to *unit remed* for remedial work if the number of correct responses is Ø or 1. If the number of correct responses is 2 or more, the program "falls through" the *x* and writes *Good work*! If there is a branch to *unit remed* (the number of correct responses is Ø or 1), *Good Work*! is not written; the instructions following the -goto- are not executed. When *unit remed* has been completed, the unit physically following *unit test1* (*unit test2*) is executed, assuming that *unit remed* does not contain a -next- instruction or another instruction that overrides this effect.

CHECK YOUR UNDERSTANDING

Section 1

Directions: Questions 1 and 2 refer to the following instruction. Choose the best answer for each question.

```
jump      count,x,one,two
```

_____ 1. If the value of *count* is 1, what happens?

a. No jump; the program "falls through" to the next instruction
b. A jump to *unit x*
c. A jump to *unit one*
d. A jump to *unit two*

_____ 2. Suppose that you don't want to change the main unit or return to instructions in the main unit following the branch. What could you use in place of -jump-?

a. `join  count,x,one,two`
b. `do  count,x,one,two`
c. `goto  count,x,one,two`
d. `do  count`

Answers

1. d. If the value of the expression is 1, the jump is to the third unit listed.
2. c. Only -goto- does not change the main unit and does not return to the instructions following the branch.

Section 2

Directions: Rewrite and execute this code using the conditional forms of -jump-, -do-, -join-, and -goto- (one at a time). Possible solutions are given in the next paragraph.

```
define  wrong    = v1
calc    wrong←Ø
unit    example
at      2Ø1Ø
write   Who is buried in Grant's tomb?
arrow   where+2
specs   bumpshift
answer  grant
endarrow
calcs   ntries-2,wrong←,wrong+1
next    wrong-1,one,two
unit    one
at      221Ø
write   unit one
unit    two
at      221Ø
write   unit two
```

The -jump-, -do-, -join-, and -goto- instructions can be used in the solution given for the preceding exercise by replacing the -next- with any one of the instructions from the list below:

```
jump    wrong-1,one,two
do      wrong-1,one,two
join    wrong-1,one,two
goto    wrong-1,one,two
```

Each of these may produce results that you may not expect. You might also add the instructions

```
at      241Ø
write   Unit example
```

following the -jump-, -do-, -join-, or -goto-. These instructions will be executed only when they follow a -do- or -join-.



6-G. CONDITIONAL INSTRUCTIONS: BRANCHING WITHIN UNITS

Identify the effects of conditional instructions that branch within units.

The -jump-, -goto-, and -do- instructions produce branches to other units. Sometimes, however, you will need to branch within a unit. You may, for example, need to branch forward, skipping some instructions, or you may want to branch back to instructions already executed. This activity discusses three conditional instructions that allow you to branch within units: -if-, -branch-, and -doto-. Before this discussion of -if-, however, a new concept must be introduced: logical expressions. Logical expressions are useful in all conditional instructions, particularly in -if-.

LOGICAL EXPRESSIONS

The expressions used with conditional instructions in preceding activities are all *arithmetic* expressions; they generate numbers used to indicate which alternatives should be performed. For example, if the number of correct answers made by a student is stored in the variable *count*, then *count* is an arithmetic expression. It can be assigned a value, such as 1 or 2. In some cases, arithmetic expressions force you to write awkward and inefficient instructions. Suppose, for example, that your program asks eight questions. You want to write *Very good* if the student answers seven or eight of the questions correctly and *You need review* if the student answers six or less correctly. If variable *count* stores the number of correct answers, you could use

```
writec  count-1$You need review$You need review$
        You need review$You need review$You need review$
        You need review$You need review$Very Good
```

With this instruction, eight alternative messages must be listed, one for each possible value of *count* (up to 7). There are only two different messages, however: *You need review* and *Very Good*. It would be more efficient to list each of these alternatives only once. In order to do this, a *logical* expression must be used instead of an arithmetic expression.

Logical expressions are either true or false in given situations. Examples are v1=1Ø, count = 72, and test <4. A logical expression could be used to simplify this code:

```
writec  count>6$
        Very Good$You need review
```

Here are some more examples of the use of logical expressions in conditional instructions:

```
answerc  count>6$
         ANSWER IF TRUE$
         ANSWER IF FALSE
calcs    count>6,var$1,2
next     count>6,true,false
jump     count>6,true,false
```

In each of these instructions the alternative performed is conditional on the truth of the expression *count* > *6*. If the expression is true, the first alternative is performed; if the expression is false, the second alternative is performed.

Conditional instructions can use logical expressions because, like arithmetic expressions, they generate numbers that are used to indicate which alternative should be performed. If a logical expression is true, –1 is generated. If a logical expression is false, ∅ is generated. A conditional instruction with a logical expression behaves like a conditional instruction with an arithmetic expression that generates only –1 or ∅.

Because logical expressions generate numbers, they can also be used in -calc- instructions. For example, the instruction

```
calc     test$(count=6)
```

stores the value –1 in the variable test if *count* = *6* is true and the value ∅ if *count* = *6* is false.

Operators that may be used in logical expressions are summarized in table 4. The table includes a definition of each operator and identification of keys used to produce each symbol.

TABLE 4
Operators in Logical Expressions

Operator	*Definition*	*Produced by*
>	Greater than	>
<	Less than	<
=	Equal to	=
≠	Not equal to	MICRO =
≤	Less than or equal to	MICRO <
≥	Greater than or equal to	MICRO >

THE -if- INSTRUCTION

The -if- instruction is really part of a family of instructions that includes -if-, -elseif-, -else-, and -endif-. Using -if- involves using a sequence of instructions from the family. The sequence must obey the following rules:

1. The -if- and -endif- instructions *must* be used in the sequence; -elseif- and -else- *may* be used.
2. The -if- instruction must be the first instruction in the sequence; -endif- must be the last.
3. The -elseif- instruction may be used more than once in the sequence; -if-, -else-, and -endif- may be used only once.
4. If -elseif- and -else- are used, all -elseif- instructions must precede the -else- instruction, and the -else- instruction must precede the -endif-.

Examples of legal sequences are shown below:

Example A

1. -if-
2. -elseif-
3. -else-
4. -endif-

Example B

1. -if-
2. -elseif-
3. -elseif-
4. -elseif-
5. -else-
6. -endif-

Example C

1. -if-
2. -elseif-
3. -endif-

Example D

1. -if-
2. -elseif-
3. -elseif-
4. -endif-

Example E

1. -if-
2. -else-
3. -endif-

Example F

1. -if-
2. -endif-

The following sequences are illegal. Rule violations are given below each sequence.

Example A

1. -if-
2. -elseif-
3. -else-

Violation:

No -endif-

Example B

1. -elseif-
2. -else-
3. -endif-

Violation:

No -if-

Example C

1. -elseif-
2. -if-
3. -endif-

Violation:

-if- is not the first instruction

Example D	*Example E*	*Example F*
1. -if-	1. -if-	1. -if-
2. -elseif-	2. -else-	2. -else-
3. -endif-	3. -else-	3. -elseif-
4. -else-	4. -endif-	4. -endif-
Violation:	*Violation:*	*Violation:*
-endif- is not the last instruction	-else- used more than once	-else- precedes -elseif-

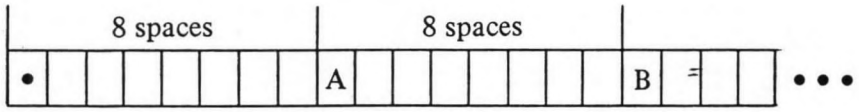
The general form for -if-, -elseif-, -else-, and -endif- is:

```
if        LOGICAL EXPRESSION
.         COMMAND TAG
elseif    LOGICAL EXPRESSION
.         COMMAND TAG
else
.         COMMAND TAG
endif
```

indented instruction

Note that -if-, -elseif-, and -else- are followed by indented instructions.

An indented instruction must begin with a period in the first space of the command field. The command of the indented instruction begins where tags usually begin—in the first space of the tag field. The tag of the indented instruction begins in the ninth space over from the first space of the command of the indented instruction. An easy way to obtain the proper spacing is to press the MICRO key and then the period.



A marks the space for the beginning of the command. *B* marks the space for the beginning of the tag.

After the indention, code is entered in the normal fashion: Type the command, press the TAB key, and type the tag. An example of an -if- sequence with indented instructions is:

```
if        score>92
.         write   Excellent
elseif    score>80
.         write   Very Good
else
.         write   You need review
endif
```

When using indented instructions, remember that indented code can only include instructions executed in the regular state. Instructions executed in the judging state *cannot* be included.

The -if- and -elseif- instructions are conditional instructions that use a logical expression in the tag. If the logical expression is true, the indented code immediately following the conditional instruction is executed. If there is no branch to a new unit in the indented code, a -jump-, for example, the -endif- instruction is then executed. If the logical expression is false, the indented code is skipped and the next unindented instruction is executed.

For example, suppose that you are writing a program that presents a quiz. You want to give the student feedback at the end of the quiz. One way to do this would be to use an -if- sequence such as:

```
if      score>92
.       write   Excellent
elseif  score>80
.       write   Very Good
elseif  score>70
.       write   Fair
elseif  score≤70
.       write   Poor
endif
```

Assume that the variable *score* holds the student's score. If the score is greater than 92 (the expression for -if- is true), the message *Excellent*! is written and the -endif- is executed. If the score is not greater than 92 (the expression for -if- is false), the indented code following -if- is skipped and the first -elseif- is executed.

Each -elseif- instruction is executed in the same manner as the -if-: When the logical expression is true, the indented code is executed and, if there is no branch in the indented code, the -endif- is executed; when the logical expression is false, the indented code is skipped and the next unindented instruction is executed.

The -else- instruction is an unconditional instruction with no tag. If the -else- instruction is executed (because the logical expressions of the preceding -if- or -elseif- instructions are false) the indented code immediately following -else- is always executed. An example of the use of -else- is:

```
if      score>92
.       write   Excellent
elseif  score>80
.       write   Very Good
```

```
elseif  score>70
.       write   Fair
else
.       write   Poor
endif
```

This example is identical to the previous example except that the -else- has replaced the instruction *elseif score* $\leq$ *70*. The effect of the two examples is the same.

Like -else-, -endif- has no tag. Unlike -if-, -elseif-, and -else-, it is not followed by indented code. Its function is to indicate the end of an -if- sequence.

CHECK YOUR UNDERSTANDING

Section 1

Directions: Questions 1 and 2 refer to the following instructions. Select the correct answers for each question.

```
if      score>90
.       write   Excellent!
elseif  score>80
.       write   Very Good
elseif  score>70
.       write   Fair
elseif  score≤70
.       write   Poor
endif
```

_____ 1. What would be written if score = 92?

a. Excellent!
b. Very good
c. Fair
d. All of the above

_____ 2. Which instruction is equivalent to the last -elseif-?

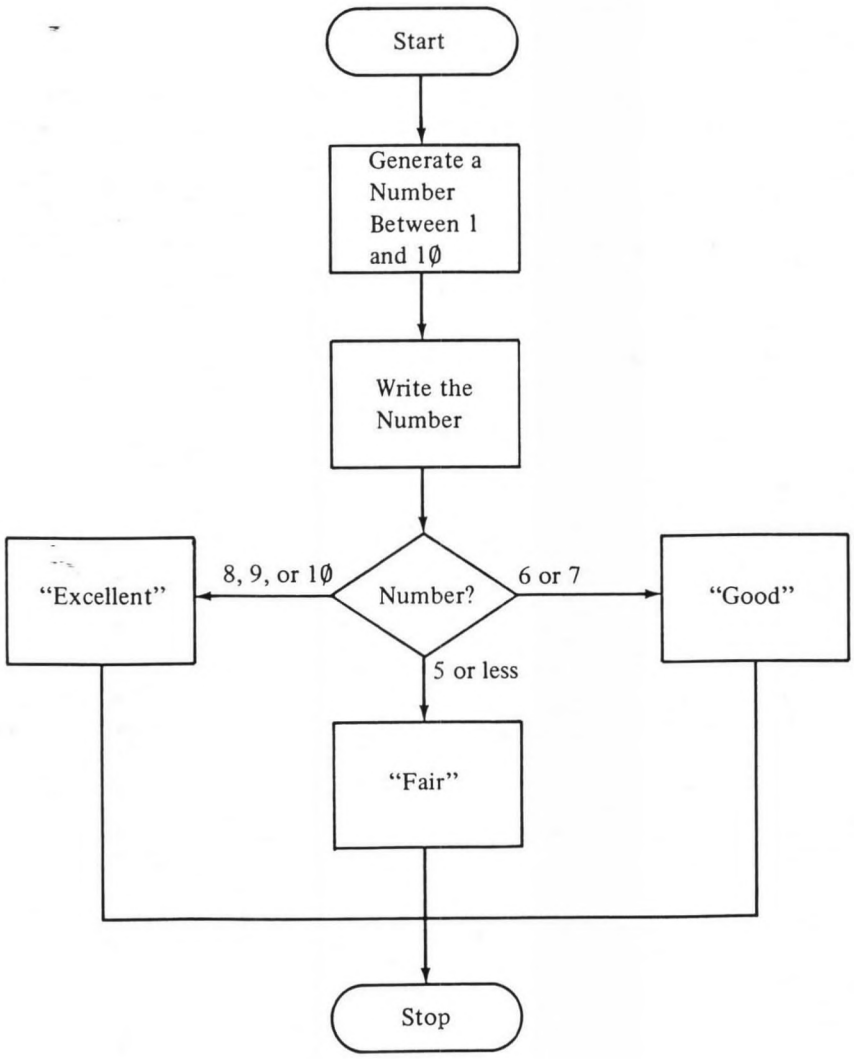
a. `elseif  score<70`
b. `else    score≤70`
c. `else`
d. `elseif`

Answers

1. a. The indented instruction following the -if- is executed, then the -endif-.
2. c. If the logical expressions of preceding -if- and -elseif- instructions are false, the indented instructions following the -else- will be executed. The -else- instruction has no tag.

Section 2

Directions: Write and execute a program that generates a number between 1 and 1Ø, writes the number, and writes *Excellent* if the number is between 8 and 1Ø, *Good* if the number is 6 or 7, and *Fair* if the number is 5 or less. Use an -if- sequence. Structure your program according to the following flowchart. One possible solution is given in the answer key.



Answer Key

```
define  score   = v1
unit    example
randu   score,10
at      405
write   The number is (s,score)
at      805
if      score≥8
.       write   Excellent
elseif  score≥6
.       write   Good
else
.       write   Fair
endif
```

THE -branch- INSTRUCTION

Branches within units can go forward, skipping some instructions, or backward to instructions already executed. The -if- sequences always branch forward. The -branch- instruction permits either forward or backward branches. The general form of the -branch- instruction is:

```
branch  EXPRESSION,STATEMENT1,STATEMENT2,STATEMENT3, ...
```

The expression can be either an arithmetic expression or a logical expression. The expression indicates to which of the given alternatives a branch is to be made. The relation of expression value to alternatives is the same as that in all other conditional instructions: with an arithmetic expression, a negative value indicates the first alternative, Ø indicates the second alternative, and so on; with a logical expression, *true* indicates the first alternative and *false* indicates the second alternative.

Alternatives are special instructions called labeled statements. A labeled statement must contain a label in the command field of the instruction. A label must begin with a one-digit number and must be no longer than seven characters. Each time you use a label in a unit, it must be given a new name. For example, *1start*, *6here*, and *9end* are all legal labels.

The function of the label is to identify the line of the program to which a branch is to be made. For example, the program segment below uses the labels *1place*, *2place*, and *3place* to identify locations:

```
        .
        .
branch  part-2,1place,2place,3place
1place
write   You've branched to 1place
```

```
goto    other
2place
write   You've branched to 2place
goto    other
3place
write   You've branched to 3place
goto    other
```

If *part* is equal to 1, 2, or 3, there is a branch to *1place*, *2place*, or *3place*, respectively. Instructions not branched to are not executed. For example, if *part* is equal to 3, the instructions following *1place* and *2place* are skipped.

A labeled statement may or may not contain a tag. If a tag is used, it must be a -calc- tag. For example, the instructions

```
        1place
        calc    count←count+1
```

are equivalent to the instruction

```
        1place  count←count+1
```

The letter *x* must be used as a space filler if a null alternative is desired. For example,

```
branch  count,x,1place,2place
```

would cause a "fall through" to the next instruction if *count* = –1.

An example illustrating the use of -branch- is shown below. To help you follow the branches, the -branch- instruction and the labeled statement are enclosed in boxes and the branches are indicated by the lines on the left side of the page. The example presents two of three possible questions in random order. Since you don't want to repeat a question, sampling without replacement must be used. This requires -setperm- and -randp-.

```
define  count    = v1
        test     = v2
calc    test←0
unit    French
setperm 3
1start  test←test+1
randp   count
at      510
writec  count-2#
        What is the English word for oui?#
        What is the English word for arret?#
        What is the English word for rue?#
arrow   where+2
answerc count-2#yes#stop#street,road
endarrow
at      510
erase   50
branch  test=2,x,1start
next    german
```

false

true

The -branch- instruction produces a branch back to *1start* if the logical expression *test = 2* is false (two questions have not been presented). If the expression is true, (two questions have been presented), the program "falls through" to the -next-.

Backward branching, sometimes called *looping*, is an extremely powerful programming technique. It allows you to use code already written to produce shorter, more efficient programs. You should use loops whenever your programs must repeat a set of identical or similar steps (like the instructions used to present a question in the last example). You must be careful in forming loops, however, to avoid three common problems:

1. Including instructions that should not be repeated
2. Failing to include instructions that should be repeated
3. Failing to provide a test that will end the loop

Figure 13 indicates how a loop can be flowcharted:

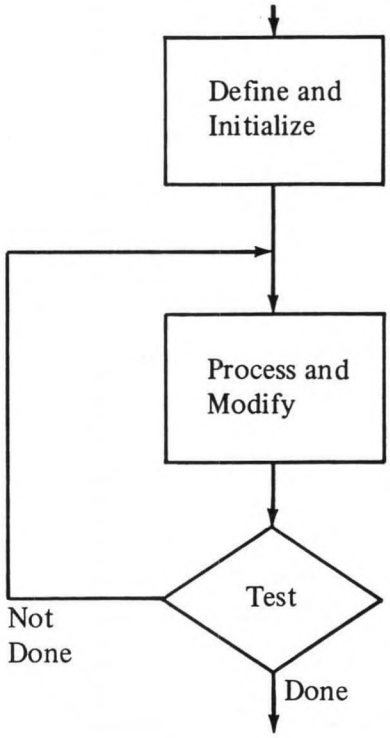


Figure 13. Flowcharting a loop

Defining and initializing should *precede* the loop. In this step, variables are defined and initialized (set to some starting value such as zero). In the preceding example, the instructions from the first -define- to the -setperm- are all part of the step *Define and Initialize*.

Process and Modify and *Test* are included in the loop. The *Process* is the set of instructions to be repeated. In this example, the instructions from the -randp- to the -erase- form the *Process*. *Modify* refers to modification of variables that are used to control the loop. These variables are usually counters that record the number of times instructions in the process have been executed. The -calc- tag used with the *1start* label in the example modifies the variable *test*, which is a counter.

Test determines whether the loop should be performed. Although the flowchart shows it following *Process and Modify*, the *Test* can be placed before, during, or after *Process and Modify*. It operates by testing the value of a variable, usually a counter. In this example, the -branch- instruction is the test: if *test = 2* is true, the loop is not performed. If *test = 2* is false, the loop is performed. The effect is to cause the instructions in the loop to be performed two times.

Probably the most common error in looping is failing to provide a test that will end a loop. An endless loop is sometimes called an *infinite loop*. In the last example, if the branch instructions had been

```
branch  test=Ø,x,1start
```

there would have been an endless loop. The expression *test* = Ø would always be false and therefore the loop would always be performed.

CHECK YOUR UNDERSTANDING

Section 1

Directions: Questions 1 through 3 refer to the following instructions. Choose the best answers for each question.

```
at      51Ø
branch  count-2,1ques,2ques,3ques
1ques
at      where+1
write   You're at 1ques
branch  count-2,x,x,2ques
goto    drill
2ques
at      where+1
write   You're at 2ques
goto    drill
3ques
at      where+1
write   You're at 3ques
branch  count-2,x,x,1ques
goto    drill
```

_____ 1. What is written if *count* = 1?

a. You're at 1ques
b. You're at 2ques
c. You're at 3ques
d. Your're at 1ques You're at 2 ques

_____ 2. What is written if *count* = 2?

a. You're at 1ques
b. You're at 2ques
c. You're at 3ques
d. Nothing

_____ 3. What is written if *count* = 3?

a. You're at 3ques
b. You're at 3ques You're at 1ques
c. You're at 3ques You're at 1ques You're at 2ques
d. Nothing

Answers

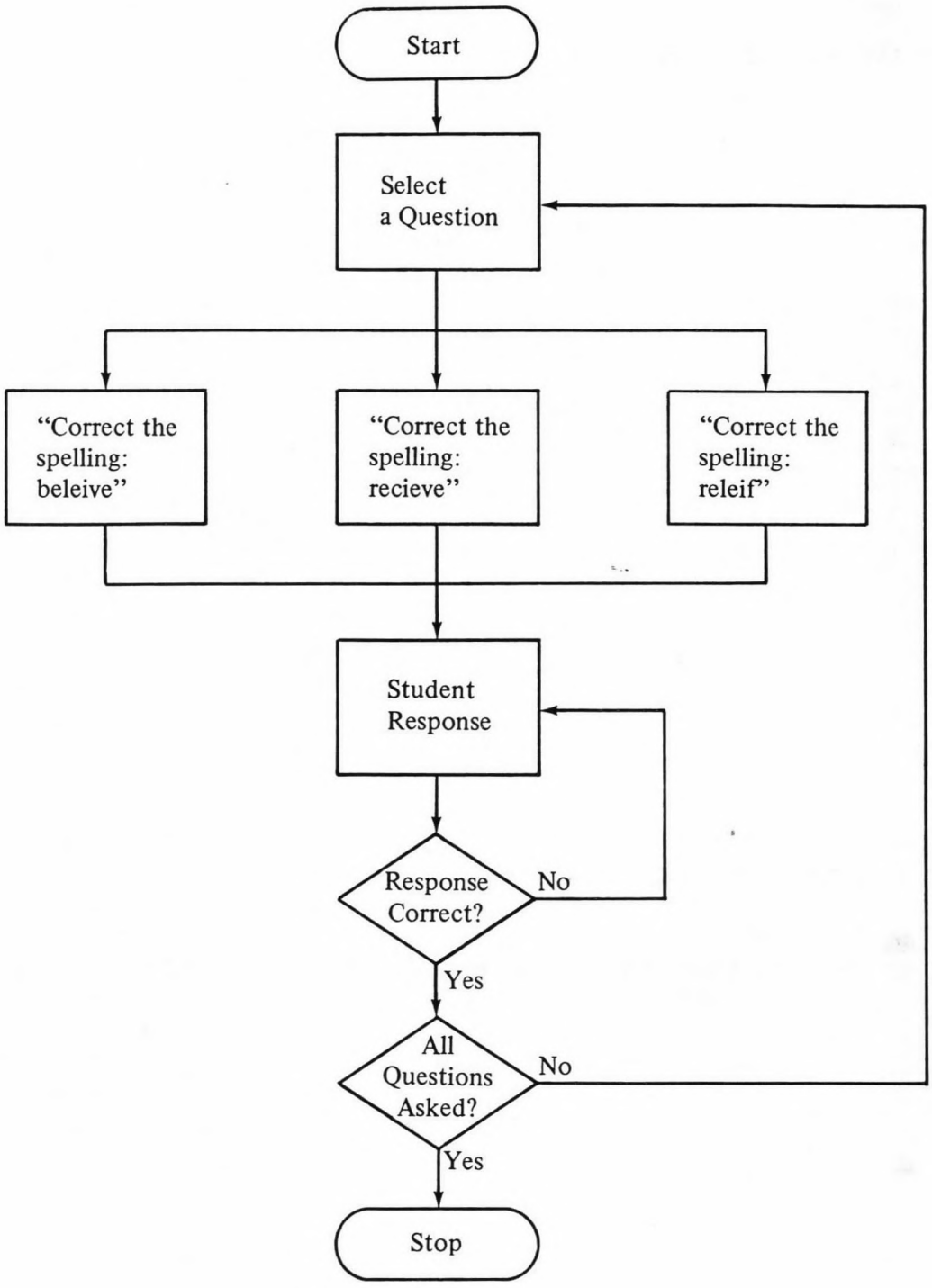
1. a. If *count* is equal to 1, *count* - 2 is equal to -1 and the first branch is to 1ques. In the second branch, *count* - 2 is still equal to -1, so the branch is ignored and the -goto- is executed. Nothing more from this unit is displayed.

2. b. If *count* is equal to 2, *count* - 2 is equal to ∅ and the branch is to 2ques. The second -write- is executed, followed by the second -goto-.

3. c. If *count* is equal to 3, *count* - 2 is equal to 1 and the first branch is to 3ques. The -branch- instruction following 3ques then causes a branch to 1ques and the -branch- instruction following 1ques causes a branch to 2ques.

Section 2

Directions: Write and execute a program that asks three questions in random order. Use -branch-, -writec-, -answerc-, -setperm-, and -randp-. Structure your program according to the following flowchart. One possible solution is given in the answer key. Save your solution for use in the next exercise.


Start
Select a Question
"Correct the spelling: beleive"
"Correct the spelling: recieve"
"Correct the spelling: releif"
Student Response
Response Correct?
No
Yes
All Questions Asked?
No
Yes
Stop

Answer Key

```
define  count    = v1
        test     = v2
calc    test←Ø
unit    example
setperm 3
1start  test←test+1
randp   count
at      4Ø5
writec  count-2*
        Correct the spelling: beleive*
        Correct the spelling: recieve*
        Correct the spelling: releif
arrow   where+2
answerc count-2*believe*receive*relief
endarrow
at      4Ø5
erase   5Ø
branch  test=3,x,1start
```

THE -doto- INSTRUCTION

The preceding example used -branch- to form a loop. The loop is *iterative* because it was performed more than once. The -doto- instruction can also be used to form iterative loops. The general form of -doto- is:

```
doto    STATEMENT,INDEX VARIABLE←INITIAL VALUE,END VALUE,
        INCREMENT
```

The statement is a labeled statement used to signify the last instruction in the loop. Unlike the labeled statements that may be used with -branch-, this statement must not have a tag. The index variable is a counter that records the number of loop iterations that have occurred. The initial value is the first value assigned to the index variable. For example, if the index variable is to be set at 1 on the first iteration, the initial value would be 1. The end value is the value the index variable is to have on the last iteration. The increment is the amount by which the index variable is to be incremented on each iteration. For example, the instruction

```
doto    1end,test←1,3,1
```

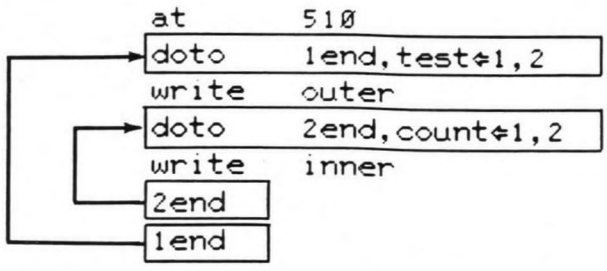
means that the loop will end at labeled statement *1end* and that there will be three iterations (the variable *test* begins with the value 1, is to end with the value 3, and is incremented by 1 on each iteration).

An example of the use of -doto- in a program is shown below. It presents two of three questions in random order. This example is identical to the -branch- example except that the -doto- has replaced the labeled statement *1start* and the labeled statement *1end* has replaced the -branch-. The effect of the two programs is the same. Note that the -doto- does not specify an increment. If no increment is specified, it is assumed to be 1.

```
define  count    = v1
        test     = v2
unit    french
setperm 3
doto    1end,test÷1,2
randp   count
at      51Ø
writec  count-2*
        What is the English word for oui?*
        What is the English word for arret?*
        What is the English word for rue?
arrow   where+2
answerc count-2*yes*stop*street,road
endarrow
at      51Ø
erase   5Ø
1end
```

Referring back to figure 13, the -doto- instruction is essentially performing the Modify and *Test*. It increments the counter (index variable) and determines whether the loop should be performed. The *Process* is all the instructions between the -doto- and the labeled statement signifying the end of the loop.

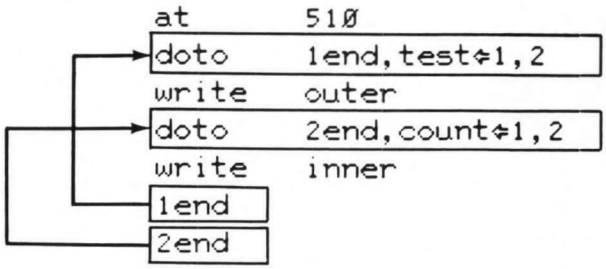
A -doto- loop can be placed within another -doto- loop. This is called *nesting*. An example is shown below. All iterations in the inner loop are performed on each iteration of the outer loop:



Execution of these instructions produces:

```
outerinnerinnerouterinnerinner
```

Nested loops must not overlap; a loop that begins within another loop must end within that loop. For example,



is illegal nesting. This is a common error.

Another error is using the same index variable in nested loops. If all iterations are to be performed, the index variables of inner loops should be different from those of outer loops. In this example, *test* and *count* were used for the two loops. If the same variable had been used for both loops, for example, *test*, execution of the instructions would produce:

```
outer↓inner↓inner
```

The second iteration of the outer loop is not performed because the end value of the index variable is reached in the inner loop.

The -branch- instruction can be used with -doto- to skip iterations by branching out of the loop or to the end of the loop. For example, suppose that you wanted to write:

```
outer↓inner↓innerouter
```

You could do this by adding a -branch- to the preceding example of legal nesting that branches around the inner loop on the last iteration of the outer loop:

```
at       510
doto     1end,test÷1,2
write    outer
branch   test=2,1end,x
doto     2end,count÷1,2
write    inner
2end
1end
```

If the value of *test* is 2, the inner loop is skipped. Execution of these instructions produces:

```
outer↓inner↓innerouter
```

CHECK YOUR UNDERSTANDING

Section 1

Directions: Questions 1 and 2 refer to the following instructions. Choose the best answer for each question.

```
at       510
doto     1end,outer⇐1,3,2
at       where+1
write    outer = ⟨s,outer⟩
branch   outer=3,1end,x
doto     2end,inner⇐1,2
at       where+1
write    inner = ⟨s,inner⟩
2end
1end
```

_____ 1. If these instructions were executed, what would be written?

a. outer = 1 inner = 1 outer = 2 inner = 2
b. outer = 1 inner = 1 inner = 2 outer = 3
c. outer = 1 inner = 1 inner = 2 outer = 2
d. Nothing, the nesting is illegal

_____ 2. If the labeled statement *2end* were placed after the labeled statement *1end*, what would be written?

a. outer = 1 inner = 1 outer = 2 inner = 2
b. outer = 1 inner = 1 inner = 2 outer = 2
c. outer = 1 inner = 1 inner = 2 outer = 2
d. Nothing, the nesting is illegal

Answers

1. b. For the first outer loop, the value of *outer* is 1 and the inner loop is performed twice. For the second outer loop, the value of *outer* is 3 and the -branch- causes the inner loop to be skipped.

2. d. The nesting is illegal because the two loops overlap.

Section 2

Directions: Rewrite this code using -doto-. One possible solution is given in the answer key.

```
define  count    = v1
        test     = v2
calc    test←Ø
unit    example
setperm 3
1start  test←test+1
randp   count
at      4Ø5
writec  count-2*
        Correct the spelling: beleive*
        Correct the spelling: recieve*
        Correct the spelling: releif
arrow   where+2
answerc count-2*believe*receive*relief
endarrow
at      4Ø5
erase   5Ø
branch  test=3,x,1start
```

Answer Key

```
define  count    = v1
        test     = v2
calc    test÷Ø
unit    example
setperm 3
doto    1end,test÷1,3
randp   count
at      4Ø5
writec  count-2♦
        Correct the spelling: beleive♦
        Correct the spelling: recieve♦
        Correct the spelling: releif
arrow   where+2
answerc count-2♦believe♦receive♦relief
endarrow
at      4Ø5
erase   5Ø
1end
```

6-H. PAUSE PROCESSING



Identify the effects of instructions used in pause processing.

The -pause- instruction is presented in unit 4. This activity returns to -pause-, discussing applications that are useful in conjunction with conditional instructions. The activity begins by reviewing the -pause- instruction and describing a -pause- format not discussed in unit 4. Two new instructions, -keytype- and -keylist-, are introduced, and the use of -pause-, -keytype-, and -keylist- with conditional instructions is discussed.

THE -pause- INSTRUCTION

Two forms of the -pause- instruction are discussed in activity 4-G. The first

```
pause    keys=KEY1,KEY2, ...
```

holds a pause until one of the keys specified in the tag is pressed. For example,

```
pause    keys=next,back
```

will hold a pause until the student presses either the NEXT key or the BACK key.

The second form is:

```
pause    SECONDS
```

With this form, the pause is held for the number of seconds specified in the tag.

The instruction

```
pause    5
```

holds the pause for 5 seconds.

There is a third form for the -pause- instruction:

```
pause    BLANK TAG
```

In this form, there is no tag. The pause is held until the student presses a key. *Any* key can be pressed, including letters and numerals.

However, the *only* effect of any key press is the break through the pause. A function key activated by a previous instruction, for example, -back-, will not have the effect indicated by the instruction; it will break the pause and the instruction following the -pause- will be executed. For example, suppose that you have the following program:

```
unit    begin
back    review
  .
  .
  .
pause
write   Good
```

You want a branch to *unit review* when the student presses BACK. If the student presses BACK at the pause, however, there will be no branch; the message *Good* will be written and instructions following the -write- will be executed.

Because of this, -pause- with a blank tag is rarely used. The tag *keys=all* is used instead. This tag has the same effect as the blank tag except that function keys activated by previous instructions will have the effects indicated by those instructions.

THE -keytype- INSTRUCTION

The -keytype- instruction allows you to determine which key the student has pressed. The general form for -keytype- is:

```
keytype VARIABLE,KEY1,KEY2,KEY3, ...
```

The tag contains a variable and a list of keys. The -keytype- instruction assigns a value to the variable specified in the tag. The value assigned is determined by the order in which keys are listed in the tag. If the student presses the key that is first in the list, the value assigned is 0. The second key assigns 1, the third key assigns 2, and so on. If the student presses a key that is not in the list, the value assigned is –1. For example, with the instruction

```
keytype press,a,b,c
```

the variable *press* is assigned the value 0 if the student presses *a*, 1 if the student presses *b*, 2 if the student presses *c*, and –1 if any other key is pressed.

You may have trouble distinguishing -keytype- from conditional instructions. The -keytype- instruction is *not* a conditional instruction. Remember that in conditional instructions the value of the variable in the tag determines which of the listed alternatives is to be performed. In -keytype-, the variable in the tag does not indicate which listed alternative is to be performed: *keys* are listed, not alternatives. The only possible action to be performed with -keytype- is to assign a value to the variable. The student's key press determines which value is to be assigned to the variable.

The -pause- instruction and -keytype- can be used in conjunction with conditional instructions whenever the alternative to be performed is conditional on a student's key press. A typical application is in building indexes. For example, the following instructions ask the student to indicate which of three options is desired. The -jump- instruction branches to *unit intro*, *unit example*, or *unit quiz*, conditional on the student's key press. If the student presses the *a* key, the value of variable *press* is 0 and the branch is to *unit intro*. If the *b* key is pressed, the value of *press* is 1 and the branch is to *unit example*. If *c* is pressed, the value of *press* is set to 2 and the branch is to *unit quiz*. If a key other than *a*, *b*, or *c* is pressed, *press* is set to -1, *press only a, b, or c* is written, and the program pauses for another key press.

```
define  press    = v1
unit    index
at      510
write   Press a, b, or c to indicate which part of
        the activity you would like to see.
        a. Introduction
        b. Examples
        c. Quiz
1start
pause   keys=all
keytype press,a,b,c
branch  press,x,1ok
at      1110
write   Press only a, b, or c
branch  1start
1ok
jump    press-1,intro,example,quiz
unit    intro
  .
  .
unit    example
  .
  .
unit    quiz
  .
  .
```

THE -keylist- INSTRUCTION

Sometimes it is desirable to assign the same value to a variable when any one of several different keys is pressed. For example, suppose that you want to test the student's knowledge of the *home* keys on a typewriter. You ask the student to press one of the home keys for the left hand (*a*, *s*, *d*, or *f*) and you need to know whether the response is correct. The -keylist- instruction can be used in conjunction with -keytype- to do this. The general form for -keylist- is:

```
keylist   GROUP NAME,KEY1,KEY2,KEY3, ...
```

The keys specified in the tag are represented by the specified group name. For example, the instructions

```
define  press    = v1
unit    type
at      51Ø
write   Press one of the home keys for the left hand.
pause   keys=all
keylist left,a,s,d,f
keylist right,j,k,l,;
keytype press,left,right
   .
   .
   .
```

would assign the value Ø to the variable *press* if any of the keys *a*, *s*, *d*, or *f* is pressed, the value 1 if any of the keys *j*, *k*, *l*, or ; is pressed, and −1 if any other key is pressed.

CHECK YOUR UNDERSTANDING

Section 1

Directions: Question 1 refers to the following instructions. Choose the best answer.

```
define  letter  = v1
unit    index
at      405
write   Press a,b,c, or d
1again
pause   keys=all
keytype letter,a,b,c,d
branch  letter,1again,1start,1end,1give,1place
```

_____ 1. If key *b* is pressed, the branch is to which labeled statement?

a. 1start
b. 1give
c. 1place
d. 1end

Question 2 refers to the following instructions. Choose the best answers.

```
define  letter  = v1
unit    index
at      405
write   Press a, b, y, or z
pause   keys=all
keytype letter,a,b,y,z
```

_____ 2. If key *z* is pressed, what is the value of variable *letter*?

a. 1
b. 2
c. 3
d. 4

Question 3 refers to the following instructions. Choose the best answer.

```
define  letter  = v1
unit    index
at      405
write   Press a, e, i, o, or u.
pause   keys=all
keylist pair1,a,e
keylist pair2,i,o
keytype letter,pair1,pair2,u
```

_____ 3. If key *i* is pressed, what is the value of variable *letter*?

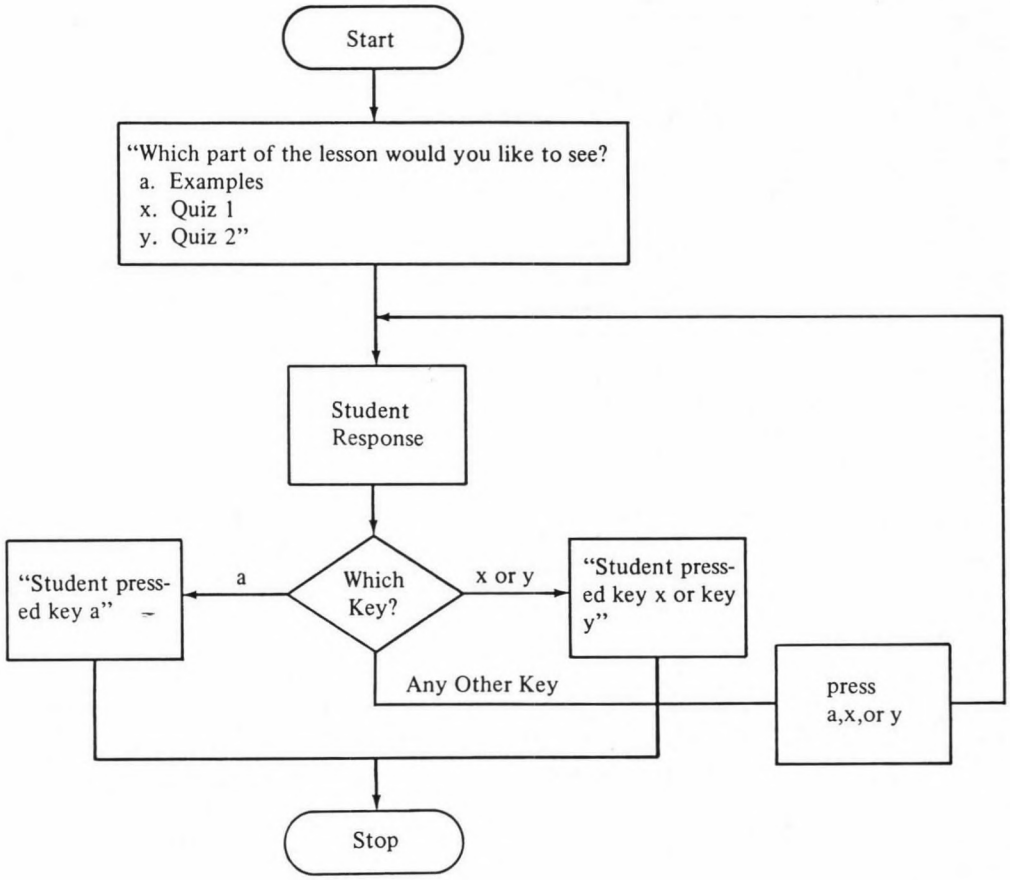
a. Ø
b. 1
c. 2
d. 3

Answers

1. d. If key *b* is pressed, the value of variable *letter* is 1, and the branch is to the third listed alternative.
2. c. The -keytype- instruction assigns the value 3 to variable *letter* if key *z* is pressed.
3. b. If either key in the group *pair2* is pressed, -keytype- assigns the value 1 to variable *letter*.

Section 2

Directions: Write and execute a program that presents an index. The student should press the keys *a*, *x*, or *y* to indicate which part of a lesson is desired. Keys *x* and *y* should be grouped. If any other key is pressed, request that the student enter one of the three keys. Following the student's keypress of *a*, *x*, or *y*, write which key is pressed, key *a* or key *x* or *y*, to be certain that the program is working correctly. Use -pause-, -keytype-, and -keylist-. Structure your program according to the following flowchart. One solution is given in the answer key.


Start
"Which part of the lesson would you like to see?
a. Examples
x. Quiz 1
y. Quiz 2"
Student Response
Which Key?
a
x or y
"Student pressed key a"
"Student pressed key x or key y"
Any Other Key
press a,x,or y
Stop

Answer Key

```
define  press   = v1
unit    index
at      405
write   Which part of the lesson would you like to see?
        a. Examples
        x. Quiz 1
        y. Quiz 2
lpause
pause   keys=all
keylist quiz,x,y
keytype press,a,quiz
at      805
writec  press*Press a, x, or y.**
branch  press,lpause,x
at      1005
writec  press-1*
        Student pressed key a*
        Student pressed key x or key y
```



6-I. PROGRAMMING WITH VARIABLES, CALCULATIONS, AND CONDITIONAL INSTRUCTIONS

Given programmable-ready material, write a program using variables, calculations, and conditional instructions.

You've learned many new instructions in this unit, and you have practiced using them individually. This activity gives you the opportunity to combine all of these in a lesson. In unit 4, you were given programmable-ready material and asked to program a lesson on classification schemes. In these two exercises, you will refine and expand that program; this will save you time because you will not have to code a new program using instructions with which you are already familiar. It will also show you how these new instructions can improve your efficiency as a programmer; in some cases, the instructions you learned in this unit will save considerable programming time.

EXERCISE 1

In this exercise, you will modify units taken from the lesson space *∅plato4*. These units have been coded to perform as specified in the following four displays from the storybook material in activity 4-F:

- Choice to try sample exercise
- Literature sample
- Train sample
- Arithmetic sample

Remember that the three samples–literature, train, and arithmetic–are somewhat repetitive. Once the student has chosen a sample, the student flow and certain aspects of the text remain constant. Differences can be found in the text for the sample problems, the correct answers, anticipated incorrect answers, feedback for correct answers, and feedback for incorrect answers.

Recode *unit try*, *unit lit*, *unit train*, and *unit math* so that you have two units: *unit try* that will calculate the variable that represents the student's choice, and *unit sample* that will execute the appropriate text and judging instructions for the student's choice.

Use a variable to represent the student's choice in *unit try* (by calculating the variable with a -calc- instruction). Substitute conditional instructions such as -writec-, -answerc-, and -wrongc- for -write-, -answer-, and -wrong- instructions in *unit sample*; this will save a considerable amount of ECS and file space. One possible solution has been included with this activity.

Code to be Modified for Exercise 1

```
unit    try
base
stop    problem
at      810
write   Once you begin the exercise, you will not be
        allowed to review.  Therefore, this is an oppor-
        tunity to practice.  Choose an area for a sample
        exercise by pressing the corresponding key.

                a. Literature
                b. Trains
                c. Arithmetic
at      3120
write   Press STOP if you do not need a sample
arrow   2010
answer  a
jump    lit
answer  b
jump    train
answer  c
jump    math
no
write   Type a, b, or c, please.
        Press NEXT and try again.
endarrow
****
unit    lit
do      lab
at      410
write   Literature into:
        English, American, French, Biography
do      quest
arrow   1110
answer  (y,yes)
wrong   (n,no)
do      wrong1
```

```
endarrow
do      quest2
arrow   2Ø1Ø
specs   bumpshift,okspell,okextra
answer  biography
at      221Ø
write   Yes, "biography" is a classification according to
        content; the others are classified by culture.
wrong   (american,english,french)
write   This categorizes literature by culture, as do
        all but one.   Press NEXT and try again.
endarrow
do      goon
***
unit    train
do      lab
at      41Ø
write   Trains into:
        Diesel, Electric, Freight, Steam
do      quest
arrow   111Ø
answer  (y,yes)
wrong   (n,no)
do      wrong1
endarrow
do      quest2
arrow   2Ø1Ø
specs   bumpshift,okspell,okextra
answer  freight
at      221Ø
write   Yes, "freight" is a classification according to
        use; the others are classified by engine type.
wrong   (steam,diesel,electric)
write   This categorizes trains by engine type, as do
        all but one.   Press NEXT and try again.
endarrow
do      goon
***
unit    math
do      lab
at      41Ø
write   Fractions into:
        Improper, Proper, Reduced
do      quest
```

```
arrow    1110
answer   (y,yes)
wrong    (n,no)
do       wrong1
endarrow
do       quest2
arrow    2010
specs    bumpshift,okspell,okextra
answer   reduced
at       2210
write    Yes, "reduced" is a classification that denotes the
         lack of a common divisor for the numerator and de-
         nominator; the others classify by relative size
         of the numerator and denominator.
wrong    (improper,proper)
write    This categorizes fractions by comparing the size
         of numerator and denominator as do all but one.
         Press NEXT and try again.
endarrow
do       goon
unit     lab
lab      try
at       3225
write    Press LAB to try another sample problem
***
unit     quest
at       810
write    Does this subdivision violate the standards of
         classification by cross-ranking?
***
unit     wrong1
write    Sorry, but one of the categories is cross-ranked.
         Press NEXT and try again.
***
unit     quest2
at       1810
write    Which subdivision is inappropriately placed in this
         classification?
***
unit     goon
at       3125
write    SHIFT-NEXT to begin exercise
next1    problem
pause    keys=lab,next1,term
```

An Acceptable Solution for Exercise 1

```
define  sample  = v1   $$identifies sample student has
*                      chosen in unit try

        prob    = v2   $$identifies the problem selected
*                      from the pool by number

        times   = v3   $$counts number of problems that
*                      student has been given

        switch  = v4   $$identifies the type of problem
*                      (whether or not it is a problem that
*                      is a bad example of classification)

        oktotal = v5   $$counts number of problems that
*                      are not violations

        nototal = v6   $$is computed from oktotal in unit
*                      summary

        rightv  = v7   $$counts number of problems that
*                      student correctly identified as
*                      either violating or not violating

        rightc  = v8   $$counts number of times that
*                      student correctly identified the
*                      offending classification on the
*                      first try.

zero    times

********
********
unit    try
base
stop    problem
at      810
write   Once you begin the exercise, you will not be
        allowed to review.  Therefore, this is an oppor-
        tunity to practice.  Choose an area for a sample
        exercise by pressing the corresponding key.

                a. Literature
                b. Trains
                c. Arithmetic
```

```
at       3120
write    Press STOP if you do not need a sample
arrow    2010
answer   a
calc     sample←1
answer   b
calc     sample←2
answer   c
calc     sample←3
no
write    Type a, b, or c, please.
         Press NEXT and try again.
endarrow
jump     sample

*        unit 'sample' replaces the units 'lit', 'train',
*        and 'math' in lesson space "@plato4".  Notice that
*        units 'lab' and 'goon' that were used in "@plato4"
*        are not used here. Before you knew how to use
*        variables, it was more efficient to code those
*        in auxiliary units because they were repeated.
*        Now, the same code is used only once, and that
*        is in this unit, 'sample'.
********
********
unit     sample
*                the following three lines of code were
*                formerly in unit 'lab'.
lab      try
at       3225
write    Press LAB to try another sample problem
*
at       410
writec   sample-2*
         Literature into:
         English, American, French, Biography*
         Trains into:
         Diesel, Electric, Freight, Steam*
         Fractions into:
         Improper, Proper, Reduced
do       quest
```

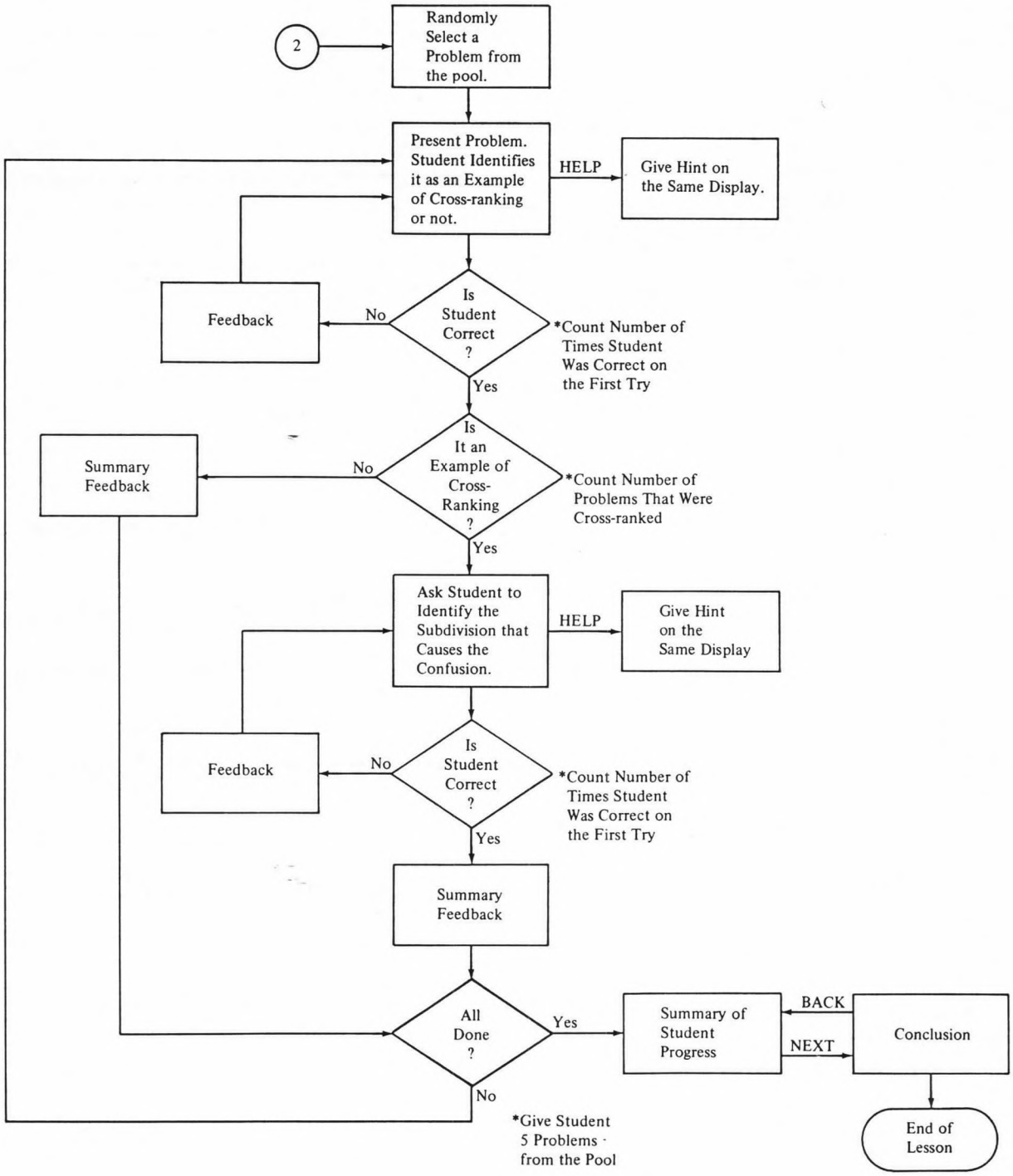
```
arrow    1110
answer   (y,yes)
wrong    (n,no)
write    Sorry, but one of the categories is cross-ranked.
         Press NEXT and try again.
endarrow
do       quest2
arrow    2010
specs    bumpshift,okspell,okextra
answerc  sample-2*biography*freight*reduced
at       2210
writec   sample-2*
         Yes, "biography" is a classification according to
         content; the others are classified by culture.*
         Yes, "freight" is a classification according to
         use; the others are classified by engine type.*
         Yes, "reduced" is a classification that denotes the
         lack of a common divisor for the numerator and de-
         nominator; the others classify by relative size
         of the numerator and denominator.
wrongc   sample-2*
         (american,english,french)*
         (steam,diesel,electric)*
         (improper,proper)
writec   sample-2*
         This categorizes literature by culture, as do
         all but one.   Press NEXT and try again.*
         This categorizes trains by engine type, as do
         all but one.   Press NEXT and try again.*
         This categorizes fractions by comparing the size
         of numerator and denominator as do all but one.
         Press NEXT and try again.
endarrow
*                the following four lines of code were
*                formerly in unit 'goon'.
at       3125
write    SHIFT-NEXT to begin exercise
next1    problem
pause    keys=lab,next1,term
***
```

EXERCISE 2

The following pages contain additional programmable-ready material for the lesson you programmed in unit 4. This exercise requires you to make coding changes at the beginning of the five problems (exercises) that are given to the student. Page 2 of the flowchart for the lesson has been modified and included in this activity; you are required to include two additional exercises in the lesson. Study the programmable-ready material provided in the following pages; then change the code of the lesson you coded in unit 4 to reflect the changes called for by this material.

Acceptable code for this exercise can be found in the file space *∅plato6*. (Enter *∅plato6* on the author mode display.) This code includes frequent comments on the instructions used; these comments are usually explanations of why parts of the lesson are coded as they are or explanations of what a particular instruction accomplishes. While completing exercise 2, feel free to examine this code at any time; you may discover ways to increase your programming efficiency.

Flowchart for Exercise 2



Storybook for Exercise 2 (Additional Problems)

Attachment for problem # 6

Text to insert at (2)

Trees into:
Cedar, Evergreen, Maple, Oak

Correct answer for question: Does this subdivision violate . . . yes

---

Help to display at (7) or (8) if key is pressed:

Three of these represent varieties of trees; the fourth is classified according to characteristics of the foliage of trees.

Correct answer for question: Which subdivision is . . . .

evergreen

Feedback to insert at (6) if student answers correctly:

Yes, "evergreen" classifies by certain foliage shedding characteristics; the others by variety.

Feedback to insert at (6) if student answers incorrectly:
(cedar, maple, oak)

This categorizes trees by variety, as do all but one. Press NEXT and try again.

Summary feedback to insert at (9) if correct answer to first question is "no":

not applicable

Attachment for problem # 7

Text to insert at (2)

Farm animals into:
Cattle, Horses, Pigs, Sheep

Correct answer for question: Does this subdivision violate . . . no

---

Help to display at (7) or (8) if key is pressed:

All of these subdivisions describe a species of farm animal. There will be no confusion when an attempt to classify is made.

Correct answer for question: Which subdivision is . . . .

not applicable

Feedback to insert at (6) if student answers correctly:

not applicable

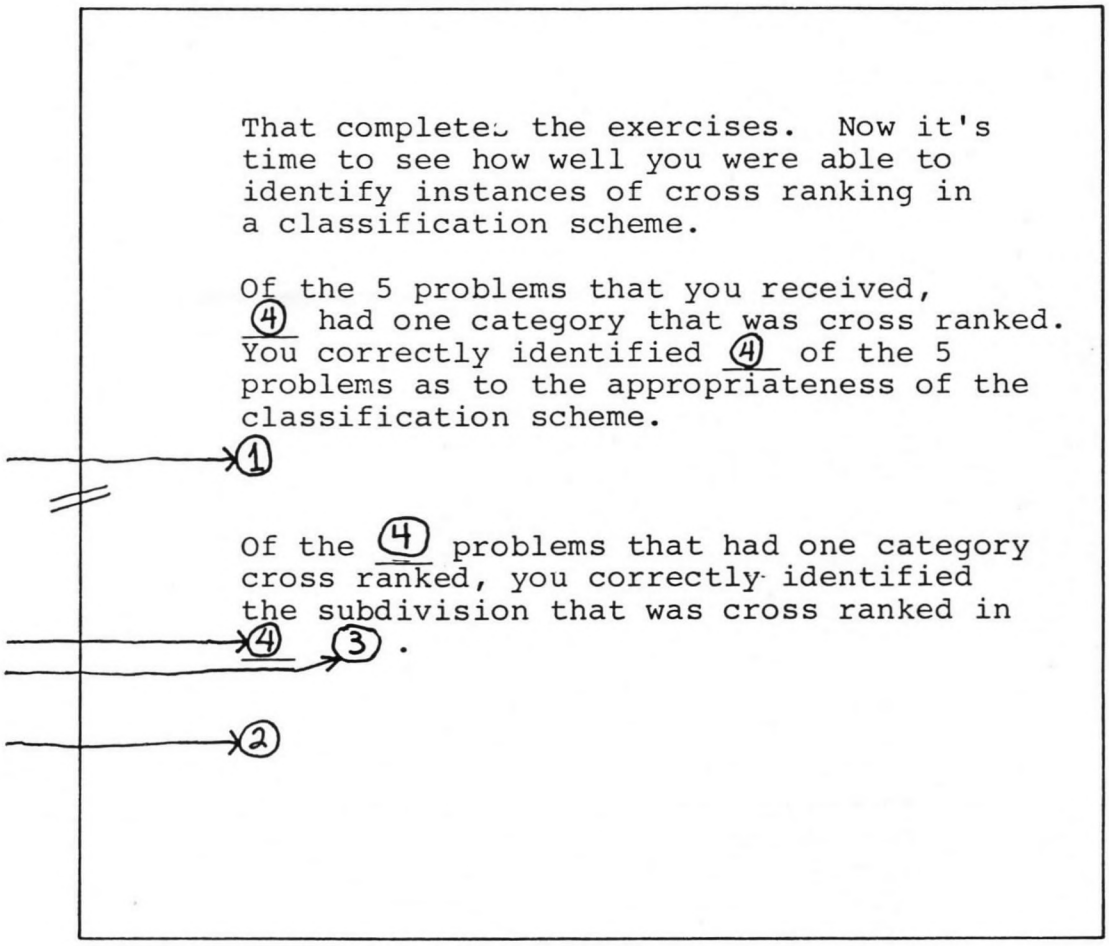
Feedback to insert at (6) if student answers incorrectly:

not applicable

Summary feedback to insert at (9) if correct answer to first question is "no":

You're right. Of course there are missing subdivisions, but that is not the question here. There are no instances of cross ranking that would cause confusion as an attempt is made to classify farm animals in these subdivisions.

Summary of student progress



Special instructions:

(1) If 5 are right: You're very good at this!
If 4 are right: You're pretty good at this.
If 3 are right: Maybe you'll do better next time.
If 2, 1, or ∅ are right: You need more practice!

(2) If all right: Very good!
If all but one are right: Pretty good!
If 2 or more are missed: Better luck next time!

(3) Use case or cases, depending on which one is appropriate.

If 1, then case.
If ∅ or >1, then cases.

(4) Insert the appropriate number here.

7

Graphics

Previous units in *PLATO Author Language: Part I* present procedures for writing words and numbers. This unit discusses instructions and other PLATO features that allow you to create displays with writing of different sizes and orientations, with lines, or with geometric figures such as squares, rectangles, circles, and triangles.

INSTRUCTIONAL OBJECTIVES

After completing this unit, you should be able to:

- Identify fine-grid positions
- Identify effects of graphics instructions
- Identify the procedures for using TERM-cursor
- Identify procedures for using the insert display (ID) and show display (SD) features of the Control Data PLATO system
- Given programmer-ready material, write a program using the graphics instructions presented in this unit

LEARNING ACTIVITIES

_____ 7-A. The Fine Grid. Text/Exercise: This activity discusses the fine grid and provides practice at locating fine-grid positions.

_____ 7-B. Graphics Instructions. Text/Exercise/CAI: Instructions used in changing the size and orientation of writing and drawing lines, rectangles, circles, and other geometric figures are discussed and applied.

_____ 7-C. TERM-cursor. Text/CAI: This activity describes capabilities of and procedures for using the TERM-cursor feature of the PLATO system.

_____ 7-D. Insert and Show Display. Text/CAI: This activity describes capabilities of and procedures for using the ID and SD features of the Control Data PLATO system.

_____ 7-E. Programming with Graphics. Text/Exercise/CAI: Programmable-ready material requiring the use of graphics instructions is presented.

TESTING DIRECTIONS

Now sign on to a Control Data PLATO terminal. If you are very familiar with graphics instructions and PLATO features used in creating graphics displays, you may wish to test out some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.

7-A. THE FINE GRID



Identify fine-grid positions.

Activity 2-B discusses the use of the coarse grid in designing screen layout. The coarse grid is adequate for the instructions you have worked with up to this point, but for instructions related to graphics, you may need to use the fine grid. This activity discusses the fine grid and gives you practice in identifying fine-grid positions.

FINE GRID AND COARSE GRID

The fine grid divides the screen into 512 rows with 512 dots in each row. The rows are numbered from bottom to top, starting with row Ø. The dots are numbered from left to right, starting with dot Ø. Fine-grid numbering is different from coarse-grid numbering. Remember that the coarse grid is divided into 32 lines with 64 spaces per line and that the lines are numbered from top to bottom starting with line 1. Like the fine-grid dots, the coarse-grid spaces are numbered from left to right, but coarse-grid numbering begins with space 1.

These differences between coarse and fine grid are summarized in figure 14.

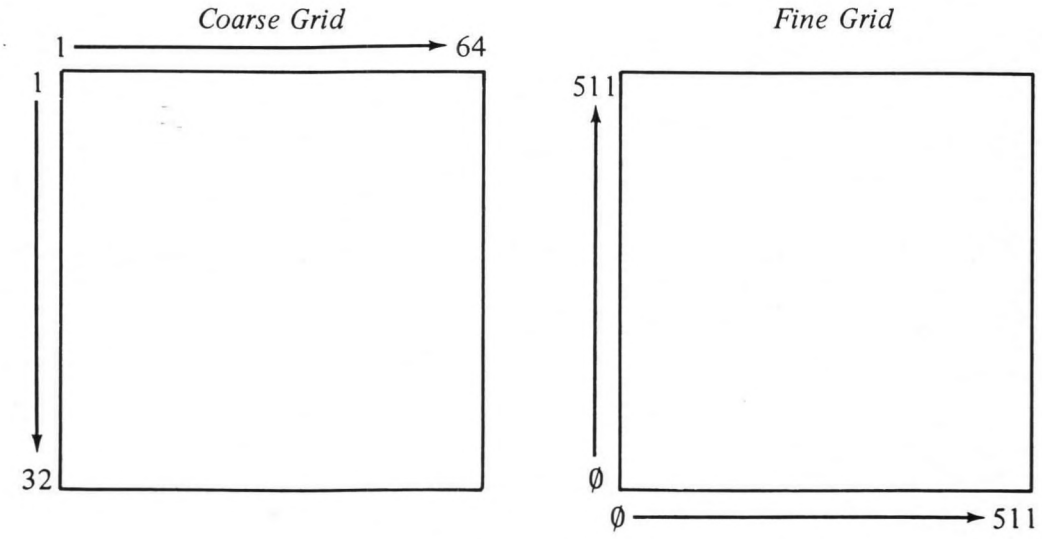


Figure 14. Coarse-grid and fine-grid numbering

As shown in figure 15, each coarse-grid location is 16 fine-grid dots high and 8 fine-grid dots wide:

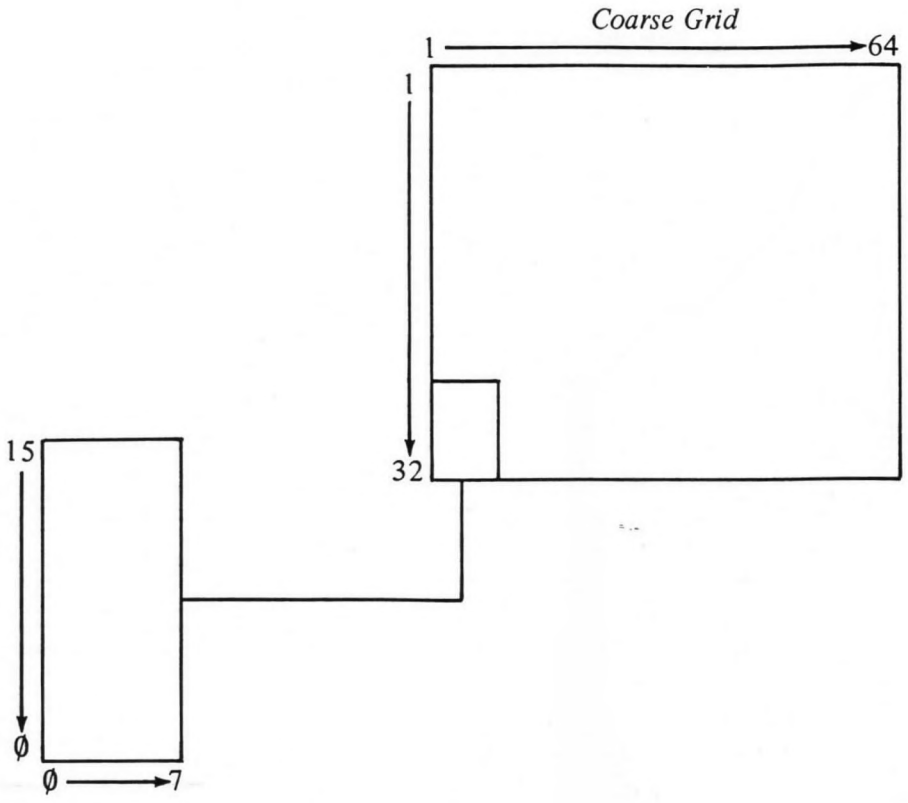
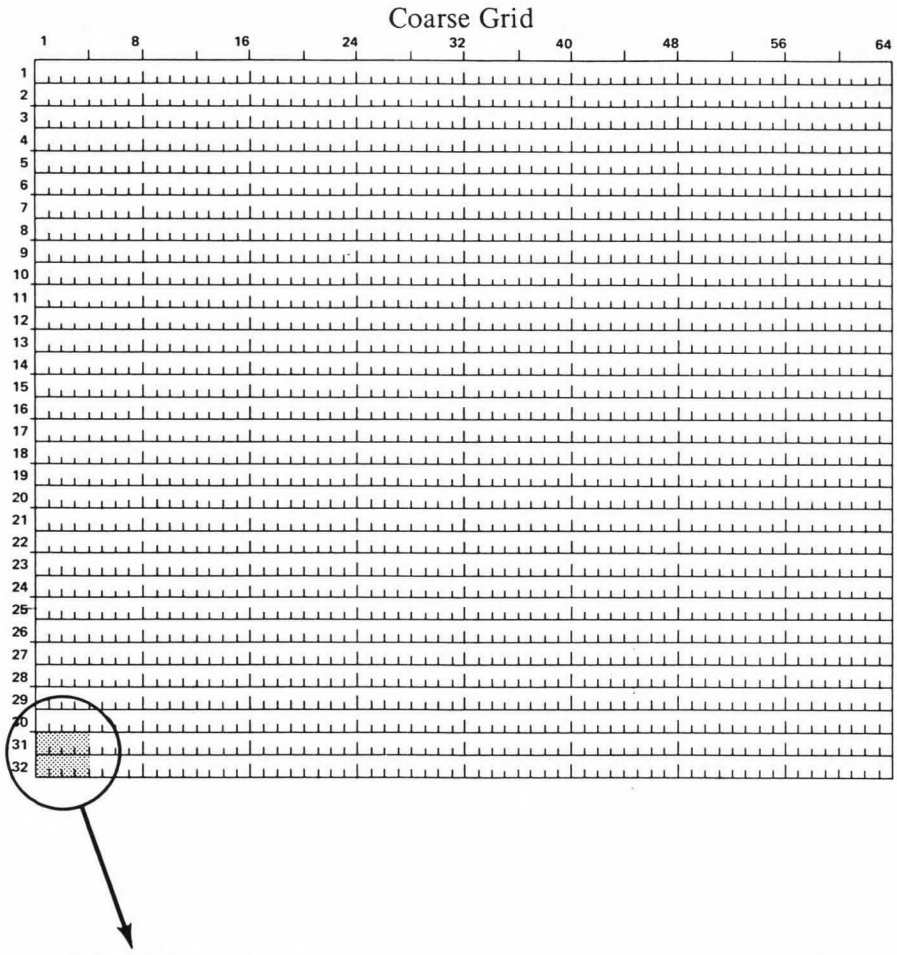


Figure 15. Coarse-grid location 32Ø1 in fine grid

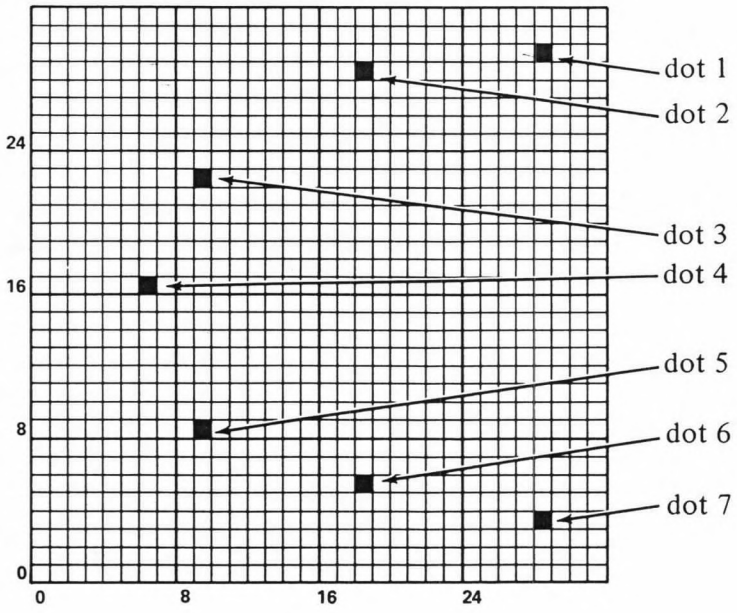
A dot on the fine grid is specified by a pair of numbers separated by a comma. The first number indicates the number of dots from the left edge of the screen. The second number indicates the number of dots from the bottom of the screen. For example, the fine-grid location Ø,511 is the dot in the upper left corner of the screen: Ø dots from the left side and 511 dots from the bottom of the screen. Position 24Ø,125 is the dot that is 24Ø dots from the left of the screen and 125 dots from the bottom.

CHECK YOUR UNDERSTANDING

Directions: The fine-grid segment corresponding to coarse-grid locations 31Ø1, 31Ø2, 31Ø3, 31Ø4, 32Ø1, 32Ø2, 32Ø3, and 32Ø4 is shown on the next page. Match each dot on the fine-grid segment with its fine-grid location.



Fine-Grid Segment
(Coarse-grid locations 31Ø1, 31Ø2, 31Ø3,
31Ø4, 32Ø1, 32Ø2, 32Ø3, 32Ø4)



_____ 1.	a. 4,28
_____ 2.	b. 6,16
_____ 3.	c. 6,18
_____ 4.	d. 8,9
_____ 5.	e. 9,8
_____ 6.	f. 9,22
_____ 7.	g. 16,6
	h. 18,5
	i. 18,28
	j. 22,9
	k. 28,3
	l. 28,18
	m. 28,29
	n. 29,28

Answers

1. m 2. i 3. f 4. b 5. e 6. h 7. k

The first number indicates the number of fine-grid dots from the left side; the second number indicates the number of fine-grid dots from the bottom.

7-B. GRAPHICS INSTRUCTIONS



Identify effects of graphics instructions.

What can you display on the screen? So far, the instructions you have for producing displays are -write-, -writec-, and -show-. These instructions permit you to display letters and numbers in one size and normal orientation (left to right, as in these sentences).

Many of the lessons you program may require fancier displays. For example, you may want to use lines or figures such as squares and circles. You may also want to increase or decrease the size of your letters and numbers. Or you may want to change their orientation so you can write like this.

This activity discusses instructions that will allow you to do all of these things. It discusses the purposes and formats of:

-size-
-rotate-
-draw-
-box-
-circle-

THE -size- INSTRUCTION

The -size- instruction allows you to increase or decrease the size of letters and numbers. There are two general forms for -size-. The first is:

```
size      NUMBER
```

The tag is a number specifying the multiple of normal size desired. For example,

```
size      2
write     text
```

would write the word *text* in twice its normal size.

The instructions

```
size    .5
write   text
```

would write the word *text* in half its normal size.

The instructions

```
size    1
write   text
```

would write the word *text* in normal size.

The second general form of the -size- instruction is:

```
size    WIDTH,HEIGHT
```

In this form, you are allowed to specify the width and height of the writing independently. For example,

```
size    2,4
```

makes the writing twice the normal width and four times the normal height.

The following is an example of how sized writing appears on screen of the PLATO terminal.

```
size Ø
```

size 2

```
size .75
```

size 2,4

size 4,2

All "sized" writing (writing with a tag other than Ø) is written more slowly than normal writing. Normal writing is written at about 18ØØ words per minute. Sized writing is written at about 6Ø words per minute.

For example, the instructions

```
size    1
write   text
```

would write the word *text* in normal size, but the speed of the writing would be slower than normal.

To return to normal writing, use the instruction:

```
size    Ø
```

For example, the instructions

```
at      5Ø5
size    2
write   text
size    Ø
write   book
```

would write *text* in twice the normal size and more slowly than normal. The word *book* would be written in normal size and speed.

CHECK YOUR UNDERSTANDING

Section 1

Directions: The following questions refer to the instructions given below. Select the correct answers for each question.

```
size    3
at      405
write   Big
size    1
write   foot
size    Ø
at      where+1
write   will get you.
```

_____ 1. What is written in normal size?

a. Big
b. foot
c. will get you.
d. None of the above

_____ 2. What is written more slowly than normal?

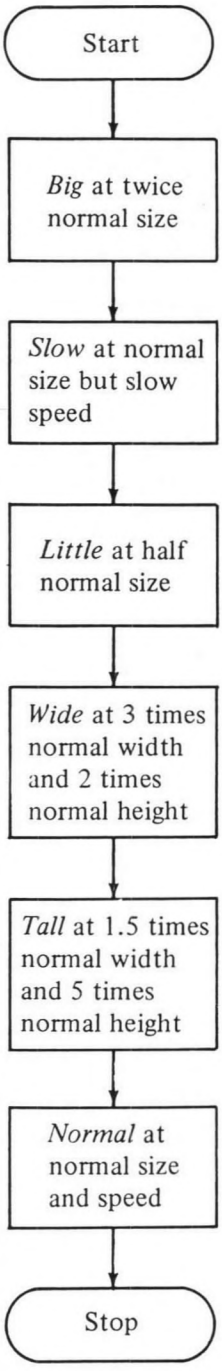
a. Big
b. foot
c. will get you.
d. None of the above

Answers

1. b, c. Writing will be in normal size if the tag of -size- is 1 or Ø.
2. a, b. All "sized" writing (writing with a tag other than Ø) is written more slowly than normal.

Section 2

Directions: Write and execute a program that writes at twice the normal size, at normal size but slower than normal speed, at half the normal size, at three times normal width and two times normal height, at 1.5 times normal width and five times normal height, and finally at normal size and speed. Use -size-. Structure your program according to the following flowchart. One possible solution is given in the answer key.



Answer Key

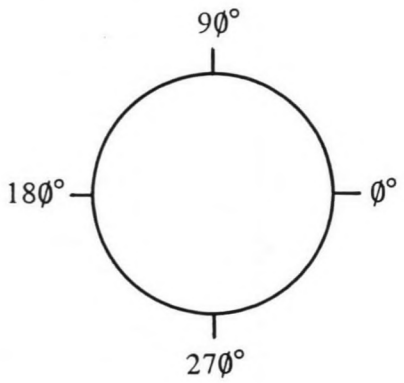
```
unit    example
size    2
at      405
write   Big
size    1
at      where+2
write   Slow
size    .5
at      where+2
write   Little
size    3,2
at      805
write   Wide
size    1.5,5
at      where+2
write   Tall
size    ø
at      where+2
write   Normal
```

THE -rotate- INSTRUCTION

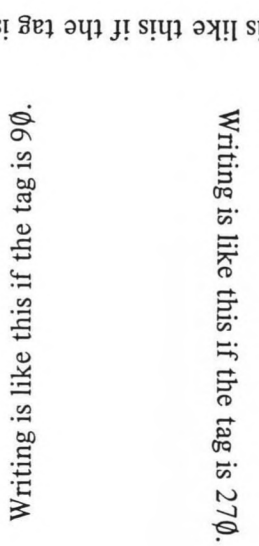
The -rotate- instruction is used to choose the orientation of letters and numbers written by instructions following the -rotate- instruction. The general form for -rotate- is:

```
rotate  DEGREES
```

The tag is a number specifing the degrees of rotation. Degrees are measured counter-clockwise from the "3 o'clock" position:



Writing is like this if the tag is Ø.



The -rotate- instruction works only with sized writing; there will be no rotation if the tag of -size- is Ø. If you want to rotate writing that is of normal size, the tag of -size- must be 1.

All writing following a -rotate- instruction will be rotated (if size is not zero). To return to the normal orientation, use

```
rotate  Ø
```

The following example shows how -size- and -rotate- can be used to label the axes of a graph:

```
at       2424
write    x axis
size     1
rotate   9Ø
at       2Ø12
write    y axis
size     Ø
rotate   Ø
```

The label *x axis* would be written horizontally beginning at coarse-grid location 2424. The label *y axis* would be written vertically beginning at coarse-grid location 2Ø12. The display would look something like this:



CHECK YOUR UNDERSTANDING

Section 1

Directions: Questions 1 and 2 refer to the following instructions. Choose the best answers for each question.

```
at       605
write    He can
size     1
rotate   90
at       where+2
write    climb
rotate   0
at       where+1
write    trees.
```

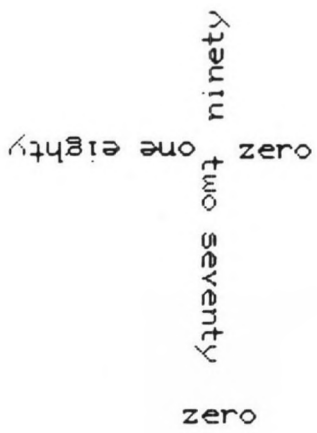
1. What is written in normal orientation?
   a. He can
   b. climb
   c. trees.
   d. None of the above
2. What is written more slowly than normal?
   a. He can
   b. climb
   c. trees.
   d. None of the above

Answers

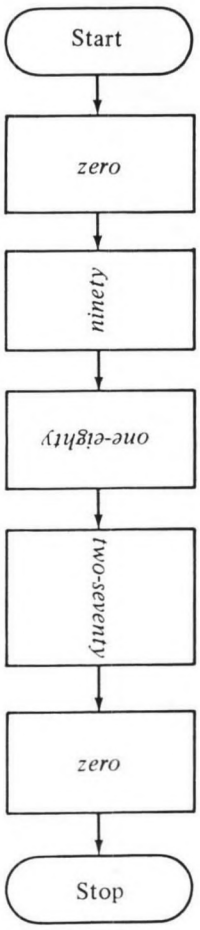
1. a,c 2. b,c

Section 2

Directions: Write and execute a program that writes the following in normal size:



Use -rotate-. Structure your program according to the following flowchart. One possible solution is given in the answer key.



Answer Key

```
unit     example
at       1030
write    zero
size     1
rotate   90
at       929
write    ninety
rotate   180
at       212,358
write    one eighty
rotate   270
at       1028
write    two seventy
rotate   0
size     0
at       1727
write    zero
```

THE -draw- INSTRUCTION

Now you know how to label the axes of a graph. But how do you draw the axes? The -draw- instruction permits you to draw straight-line segments. These straight-line segments could be the axes of graphs. They could also be the lines that *underline* important words; or they could be the sides of a geometric figure, like a triangle. The general form for -draw- is:

```
draw     LOCATION 1;LOCATION 2;LOCATION 3; ...
```

The tag contains a list of locations specifying the beginning and end points of lines to be drawn. The first line is drawn from location 1 to location 2, the next line is drawn from location 2 to location 3, and so on. The locations are separated by semicolons.

The locations are either coarse-grid or fine-grid locations. Both types of locations can be used in the same -draw- instruction. For example,

```
draw     2415;384,128;849;2415
```

uses coarse grid to specify the first, third, and fourth locations and fine grid to specify the second location. This instruction causes the drawing of a large triangle.

Fine grid can always be used instead of coarse grid, but its greatest value is in creating graphics requiring precise positioning. The following instruction, for

example, draws a triangle within a single coarse-grid location (2449):

```
draw    384,128;390,128;390,138;384,128
```

Using coarse-grid locations with -draw- can cause difficulties in predicting exactly where the line is to be drawn. Remember that a coarse-grid location is really 16 fine-grid dots high and 8 fine-grid dots wide. Where in the fine grid does the line begin or end when a coarse-grid location is specified? The point referenced by a coarse-grid location is always the lower left corner of the fine-grid representation for the location. For example, the coarse-grid location 2449 has the fine-grid representation shown in figure 15:

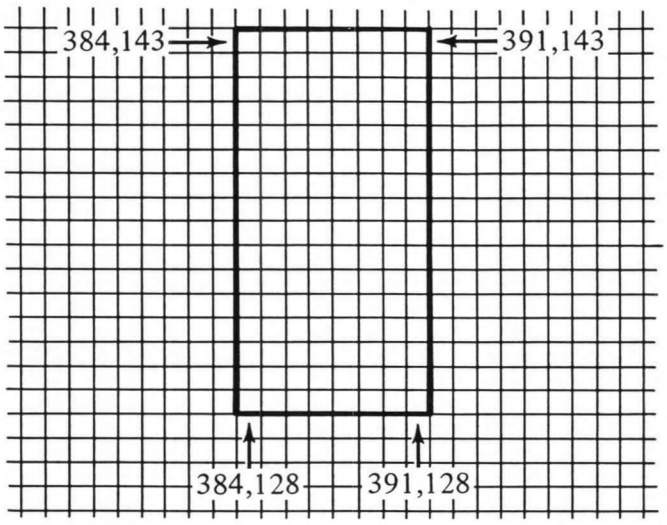


Figure 15. Fine-grid representation of coarse-grid location 2449

A line beginning at coarse-grid location 2449 would begin at fine-grid location 384,128. A line ending at coarse-grid location 2449 would end at fine-grid location 384,128.

One common area of difficulty in predicting where a line is to be drawn is in underlining. Suppose that you want to write x axis. The PLATO system places letters written at a location so that the bottom of the letter is several dots above the lower edge of the fine grid. Because of this, you can start your line at the same coarse-grid location used for the location of the first letter.

Thus, if you use

```
at      2424
write   x axis
```

the first location specified in the -draw- instruction could be 2424.

To determine the location of the end of the line, count the number of characters (including spaces) to be underlined and add this number to the beginning location.

Since there are six characters in *x axis*, the end location would be 243Ø and the appropriate -draw- instruction would be:

```
draw    2424;243Ø
```

Note that the *s* in *x axis* is written at location 2429. The line must end at 243Ø, however, because the actual end of the line is at the lower left corner of location 243Ø. The instruction

```
draw    2424;2429
```

would produce something like this:

x axis

There is an easier way to do this *if* the line is drawn immediately after *x axis* is written. In this case, the value of the system variable *where* will be the coarse-grid location following the location of the *s* in *x axis*. The instructions

```
at      2424
write   x axis
draw    where;2424
```

will write *x axis* and draw a line from the lower left corner of the location following the *s* (location 243Ø) to the lower left corner of location 2424.

You can draw discontinuous lines with the -draw- instruction. Suppose that you wanted to underline two words in a sentence:

The French word for yes is oui.

One way to do this would be to use two -draw- instructions.

```
at      4Ø5
write   The French word for yes is oui.
draw    425;428
draw    432;435
```

A more efficient way is to use a single -draw- that treats the underlinings as part of a discontinuous line segment. The word *skip* is used to signify the discontinuity. The instruction

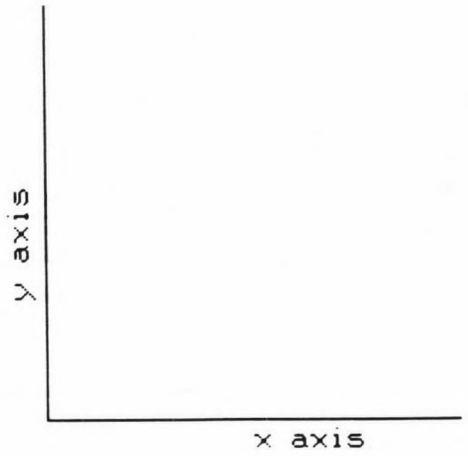
```
draw    425;428;skip;432;435
```

is equivalent to the two -draw- instructions above.

As a final example, look at the axes that were labeled using the -size- and -rotate- instructions:

```
at        2424
write     x axis
size      1
rotate    90
at        2012
write     y axis
size      0
rotate    0
draw      1213;2313;2335
```

These instructions produce a display that looks something like this:



CHECK YOUR UNDERSTANDING

Section 1

Directions: Questions 1 and 2 refer to the following instructions. Select the correct answers for each question.

```
at      605
write   He can climb trees.
```

_____ 1. Which instruction would underline *can*?

a.
```
draw    608;611
```
b.
```
draw    608;610
```
c.
```
draw    609;612
```
d.
```
draw    611;608
```

_____ 2. Which instruction would underline everything but *can*?

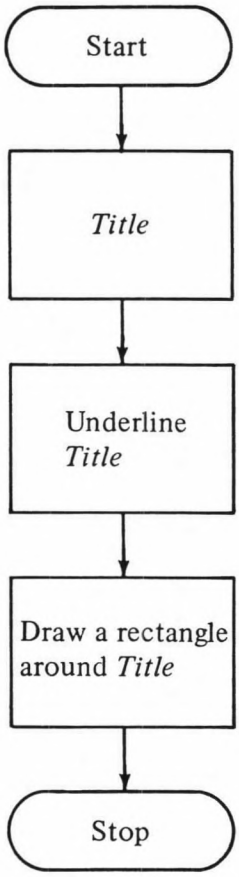
a.
```
draw    605;607;skip;612;623
```
b.
```
draw    605;607
draw    612;623
```
c.
```
draw    605;606;skip;612;622
```
d.
```
draw    605;606;612;622
```

Answers

1. a,d. The line should be drawn between the coarse-grid location of the *c* and the coarse-grid location following the *n*.
2. a,b. One -draw- instruction with a *skip* or two -draw- instructions can be used.

Section 2

Directions: Write and execute a program that writes the word *Title*, underlines it, and draws a rectangular border around it. Use -draw-. Structure your program according to the following flowchart. One possible solution is given in the answer key. Save your program for the next exercise.


Start
Title
Underline *Title*
Draw a rectangle around *Title*
Stop

Answer Key

```
unit    title
at      232,280
write   Title
draw    232,280;272,280
draw    1329;1336;1536;1529;1329
```

THE -box- INSTRUCTION

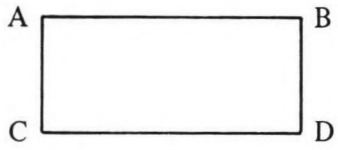
The -draw- instruction can be used to create any geometric figure. You have already seen an example that produces a triangle. Squares and rectangles can also be created using -draw-. For example, this instruction draws a rectangle:

```
draw    2420;2440;840;820;2420
```

Rectangles and squares are common figures in CAI lessons. They are often used as borders to frame parts of a display, particularly on the title display. Because they are such common figures, a special instruction, the -box- instruction, is available. The general form for -box- is:

```
box     CORNER 1;CORNER 2;THICKNESS
```

The tag specifies the locations of two corners of the box and the thickness of the border. The corners must be *opposite* corners. For example, in the box below, the locations of corners *A* and *D* or corners *B* and *C* could be specified.



The thickness of the border is the number of lines used in the border. For example,

```
box     1616;2448;2
```

draws a box with opposite corners at coarse-grid locations 1616 and 2448 and a border 2 lines thick.

The number specifying the thickness can be positive or negative. If the number is positive, the lines are added from the inside to the outside. That is, the inner line is drawn first and the outer line is drawn last. If the number is negative, the lines are added from the outside in; the outer line is drawn first and the inner line is drawn last. The first line drawn is the line specified by the corners in the -box- instruction.

If the thickness is specified as -1, Ø, or 1, or if the thickness is not specified, the thickness will be one line.

The location of the corners can be either coarse-grid or fine-grid locations. The following is an example of an instruction using fine-grid location:

```
box      120,240;380,150;2
```

CHECK YOUR UNDERSTANDING

Section 1

Directions: Questions 1 and 2 refer to the following instruction. Choose the best answer for each question.

```
box      2016;2456;-2
```

_____ 1. What kind of figure would be produced?

a. A square
b. A rectangle, wider than it is tall
c. A rectangle, taller than it is wide
d. Two lines drawn from location 2016 to location 2456

_____ 2. How would the lines be drawn?

a. Inner line first, then outer line
b. Both lines appear simultaneously
c. Outer line first, then inner line
d. None of the above

Answers

1. b 2. c

Section 2

Directions: Rewrite this -draw- instruction (from the preceding example) so that -box- is used to draw the rectangle:

```
draw     1329;1336;1536;1529;1329
```

One possible solution is given in the answer key. Try using different thicknesses. Save your program for the next exercise.

Answer Key

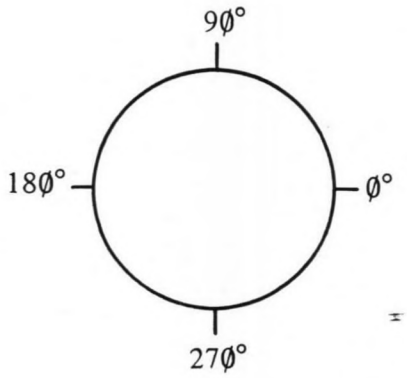
```
unit    title
at      232,28Ø
write   Title
draw    232,28Ø;272,28Ø
box     1329;1536;2
```

THE -circle- INSTRUCTION

Circles can be produced using the -draw- instruction to draw a series of very short line segments. There is an easier method: the -circle- instruction. The general form of -circle- is:

```
circle  RADIUS,DEGREE 1,DEGREE 2
```

The tag specifies the radius of the circle in fine-grid dots and the beginning and ending degrees. The radius is measured from a location specified by a preceding -at- instruction. The tag of the -at- may be either a coarse-grid or a fine-grid location. As with -rotate-, degrees are measured counter-clockwise from the "3 o'clock" position:



For example, the instructions

```
at      1632
circle  1ØØ,Ø,18Ø
```

would draw a circle centered at coarse-grid location 1632 with a radius of 1ØØ fine-grid dots and a shape like this:



If the tag contains only the radius (no beginning and ending degrees are specified), a full circle is drawn.

CHECK YOUR UNDERSTANDING

Section 1

Directions: The following questions refer to the instructions given below. Choose the best answer.

```
at       1632
circle   9Ø
```

_____ 1. What would be drawn?

a. A 9Ø-degree arc of a circle
b. A circle with a 9Ø-dot diameter
c. A circle with a 9Ø-dot radius
d. A circle with a 9Ø-dot circumference

_____ 2. Which instruction would replace the -circle- instruction above if you wanted to draw a half-circle that is open at the bottom?

a. `circle  9Ø,Ø,18Ø`
b. `circle  9Ø,9Ø,27Ø`
c. `circle  9Ø,18Ø,36Ø`
d. `circle  9Ø,27Ø,9Ø`

Answers

1. c. The tag specifies the number of dots in the *radius*.
2. a. The half-circle should be drawn from Ø degrees to 18Ø degrees.

Section 2

Directions: Rewrite this code from the preceding example so that a circle is drawn around the title instead of a box. Try to make different line thicknesses using repeated executions of -circle-. One possible solution is given in the answer key.

```
unit    title
at      232,280
write   Title
draw    232,280;272,280
box     1329;1536;2
```

Answer Key

```
unit    title
at      232,280
write   Title
draw    232,280;272,280
at      252,288
circle  28
circle  29
```

(The second -circle- instruction adds a one-line thickness.)

7-C. TERM-CURSOR



Identify the procedures for using TERM-cursor.

With practice, you'll get better at locating coarse-grid and fine-grid positions. No matter how much practice you have, however, you'll never be able to locate grid positions with complete accuracy just by looking.

CAI INTRODUCTION

This activity discusses a PLATO system feature that allows you to locate fine- and coarse-grid positions of any point on the screen: TERM-cursor.

Now sign on to a Control Data PLATO terminal using your *student* sign-on. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description page will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "TERM-cursor," press LAB to take the lesson. When you have completed this activity, read the following summary.

SUMMARY

TERM-cursor can be used in author mode from anywhere on the PLATO system. To use TERM-cursor, follow these steps:

1. Press the TERM (SHIFT-ANS) key.
2. When the system asks *what term?*, enter *cursor*.
3. To move the cursor, press the *arrow* keys (*d*, *e*, *w*, *q*, *a*, *z*, *x*, and *c*). To move the cursor more rapidly, shift the arrow keys (*D*, *E*, *W*, *Q*, *A*, *Z*, *X*, and *C*).
4. To move the cursor in fine grid, press the *f* key; to move in coarse grid, press the *g* (for gross) key.
5. To find the coarse- and fine-grid position of the cursor after you have moved it, press NEXT. The position is then updated in the lower right-hand corner of the display.
6. To leave TERM-cursor, press BACK.

7-D. INSERT AND SHOW DISPLAY



Identify procedures for using the insert display (ID) and show display (SD) features of the Control Data PLATO system.

Creating graphics displays can be difficult and time consuming, especially if the graphics are complex. Part of the difficulty is in testing the graphics you're creating; condensing takes time, both yours and the system's.

CAI INTRODUCTION

This activity discusses two PLATO system features that assist you in creating and testing graphics displays. The insert display (ID) feature helps you create displays. The show display (SD) feature helps you test displays. The activity has five parts:

- Introduction and objective
- Getting into ID mode
- Drawing the perfect square
- Showing displays with SD
- Other ID options

You should take all of the parts, beginning with the introduction.

The activity recommends two Control Data PLATO lessons as prerequisites: *Øtermcurso* and *Øediting1*. You may already have seen both lessons, *Øtermcurso* in activity 7-C and *Øediting1* in activity 3-D.

Now sign on to a Control Data PLATO terminal using your *student* sign-on. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description page will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "An Introduction to the Display Generation Options in the TUTOR Editor," press LAB to take the lesson. When you have completed this activity, read the following summary.

SUMMARY

The ID option allows you to insert a display. To use ID, you must be on the line display page and follow these steps:

1. To insert a display, type *id* and the line number after which the display is to be inserted, for example, *id4*, and then press NEXT. If no line number is specified (you type only *id*), *id1* is assumed.
2. To move the cursor, press the *arrow* keys (*d*, *e*, *w*, *q*, *a*, *z*, *x*, and *c*) or the shifted *arrow* keys (*D*, *E*, *W*, *Q*, *A*, *Z*, *X*, and *C*).
3. To move the cursor in the fine grid, press the *f* key. To move in coarse grid, press *g* (for gross).
4. To find the coarse- and fine-grid position of the cursor, press NEXT or the SPACE bar.
5. To draw lines, place the cursor at the beginning of the line and press the *p* key. Then move the cursor to the end of the line and press *l*.
6. To skip lines, move the cursor to the start of the new line and press the *p* key.
7. To add text to the display, type *t* for the text option. To change the size of your text, type *S* (shifted *s*) for the size option. To change the orientation of your text, type *R* (shifted *r*) for the rotate option.
8. To leave ID, press SHIFT-BACK.

The SD option allows you to show a display. To use SD, you must be on the line display page and follow these steps:

1. To show a display, type *sd*, then the number of lines to show, and then NEXT.
2. If you get an error message for a non-graphics instruction, ignore it. Press NEXT until you reach a graphics instruction.
3. When you see the display, you will be in ID. You can insert a display or leave by pressing SHIFT-BACK.

7-E. PROGRAMMING WITH GRAPHICS



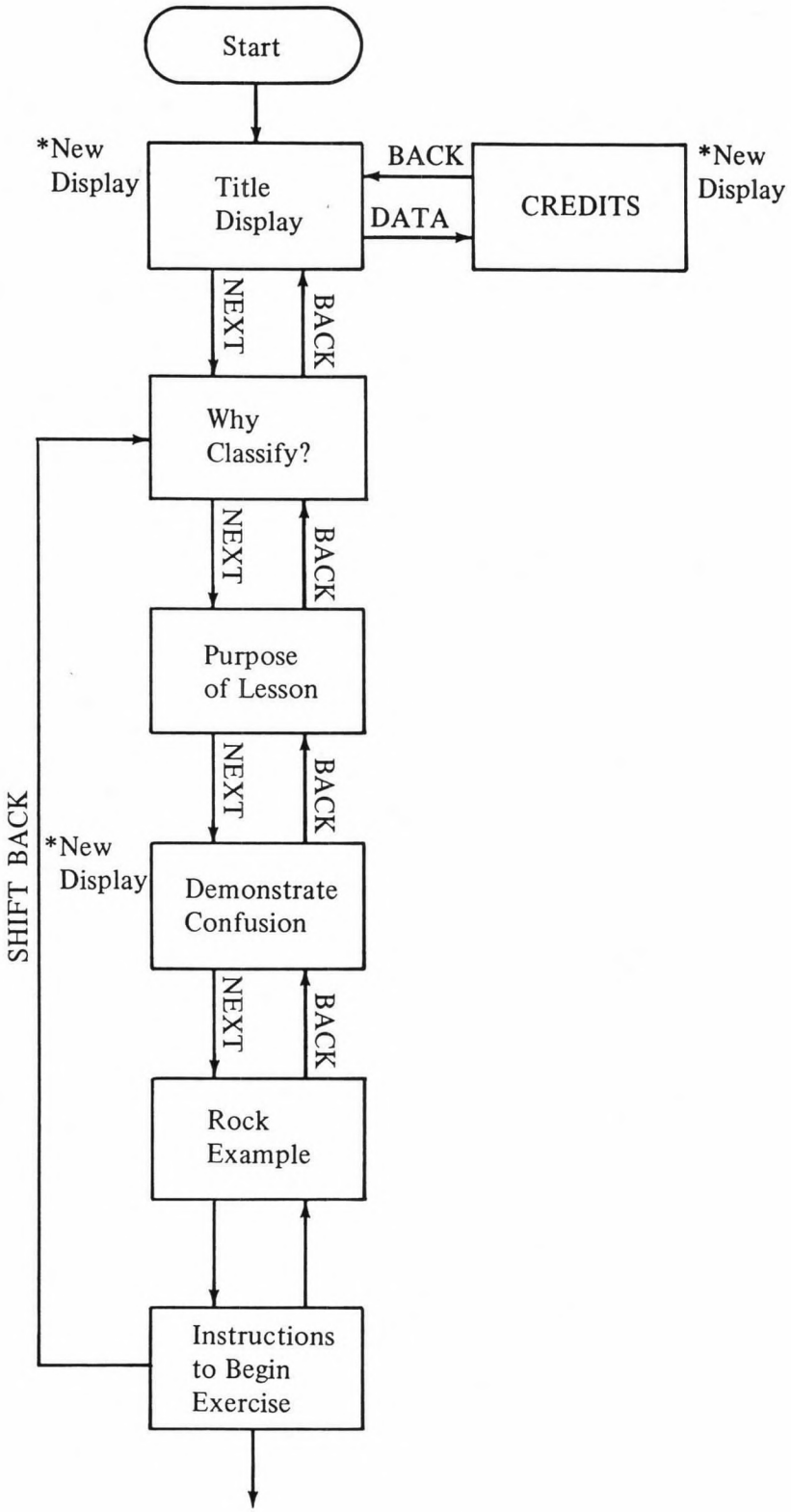
Given programmable-ready material, write a program using the graphics instructions presented in this unit.

In this unit you have learned several instructions that you can use to create graphics displays. Now you will use these instructions to add the final displays to the classification lesson that you coded in units 4 and 6.

The following pages contain additional programmable-ready material and a revised flowchart. They require you to:

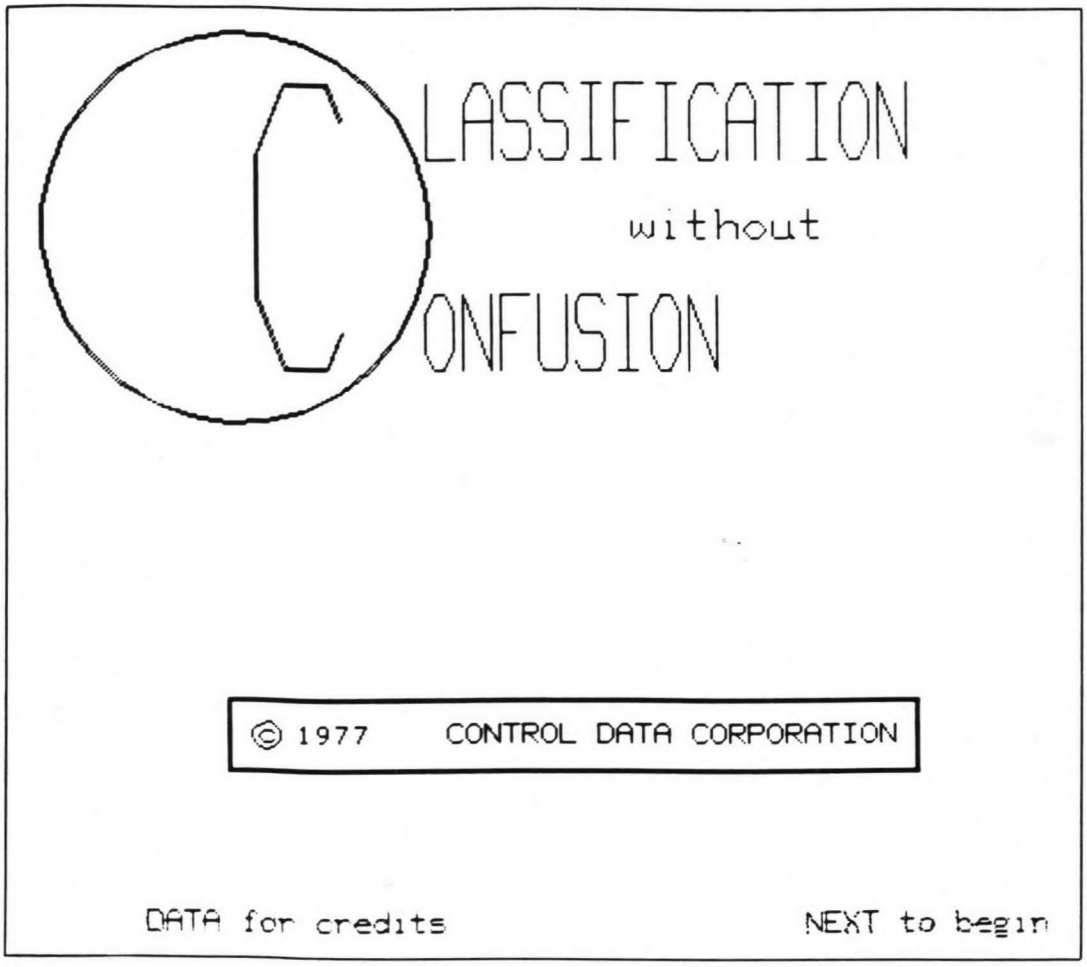
- Insert a title display
- Insert a credits display
- Insert a display demonstrating a confusing classification scheme
- Underline several important phrases
- Draw a box around each problem

Acceptable code for this activity can be found in the file space *∅plato7*. While completing this activity, feel free to examine this code and the comments at any time.



This is the end of the modifications to the flowchart introduced in unit 4 and revised in unit 6.

Title display



Special instructions:

Make the letters in LASSIFICATION and ONFUSION tall and narrow. Make the C very tall.

Make the circle around "C" and the box around the copyright heavier than one thickness.

Credits

A CAI activity designed and developed for

PLATO Author Language: Part I

CREATE curriculum

Development Organization .. Courseware Development Division

Designed and Developed by .. Molly Reko

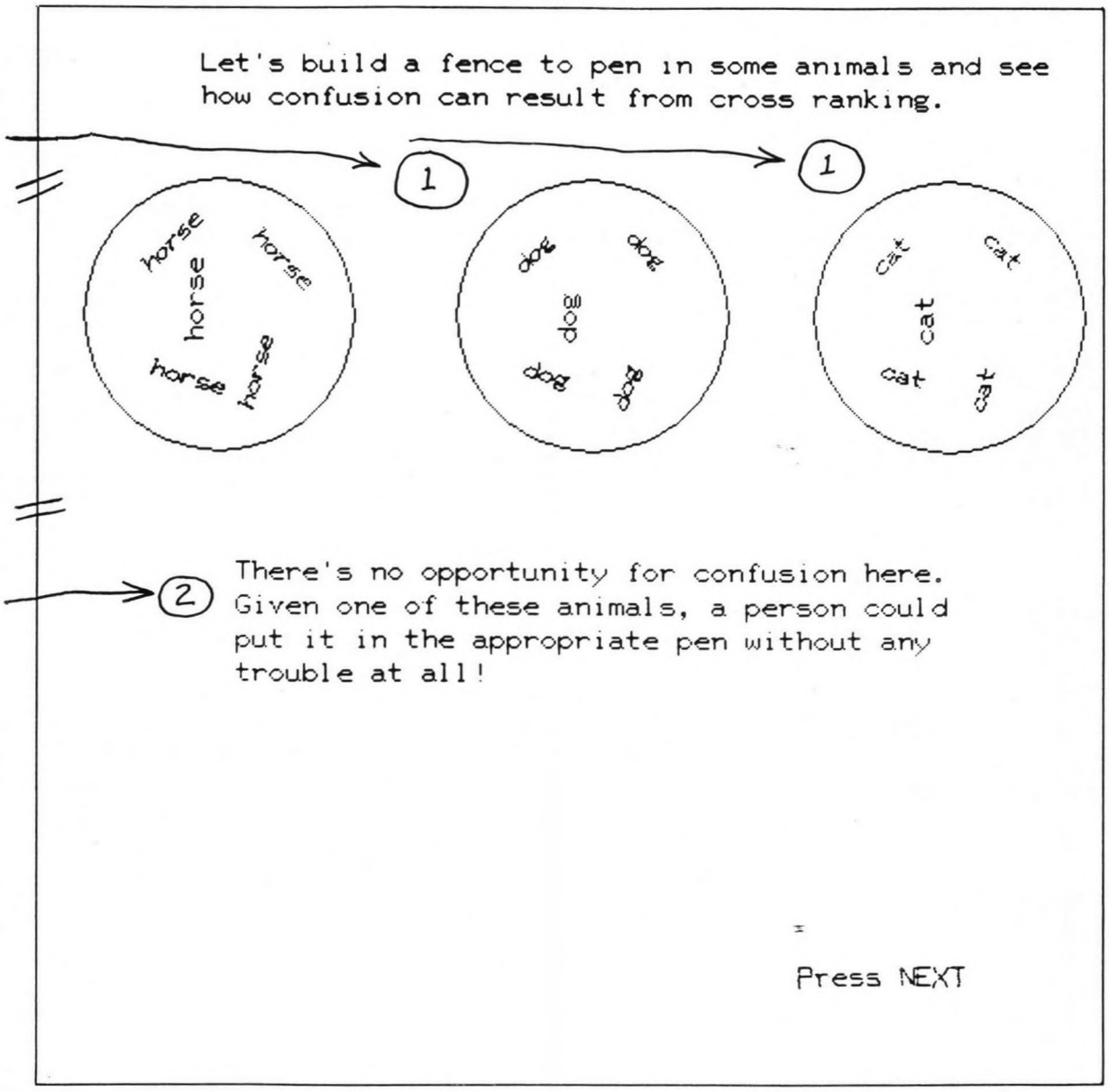
Programmed by.. (insert your name)

Module developed by .. Karen Brown
William Bewley

This activity has been programmed using instructions introduced in PLATO Author Language: Part I.

NB

Demonstrate confusion
A

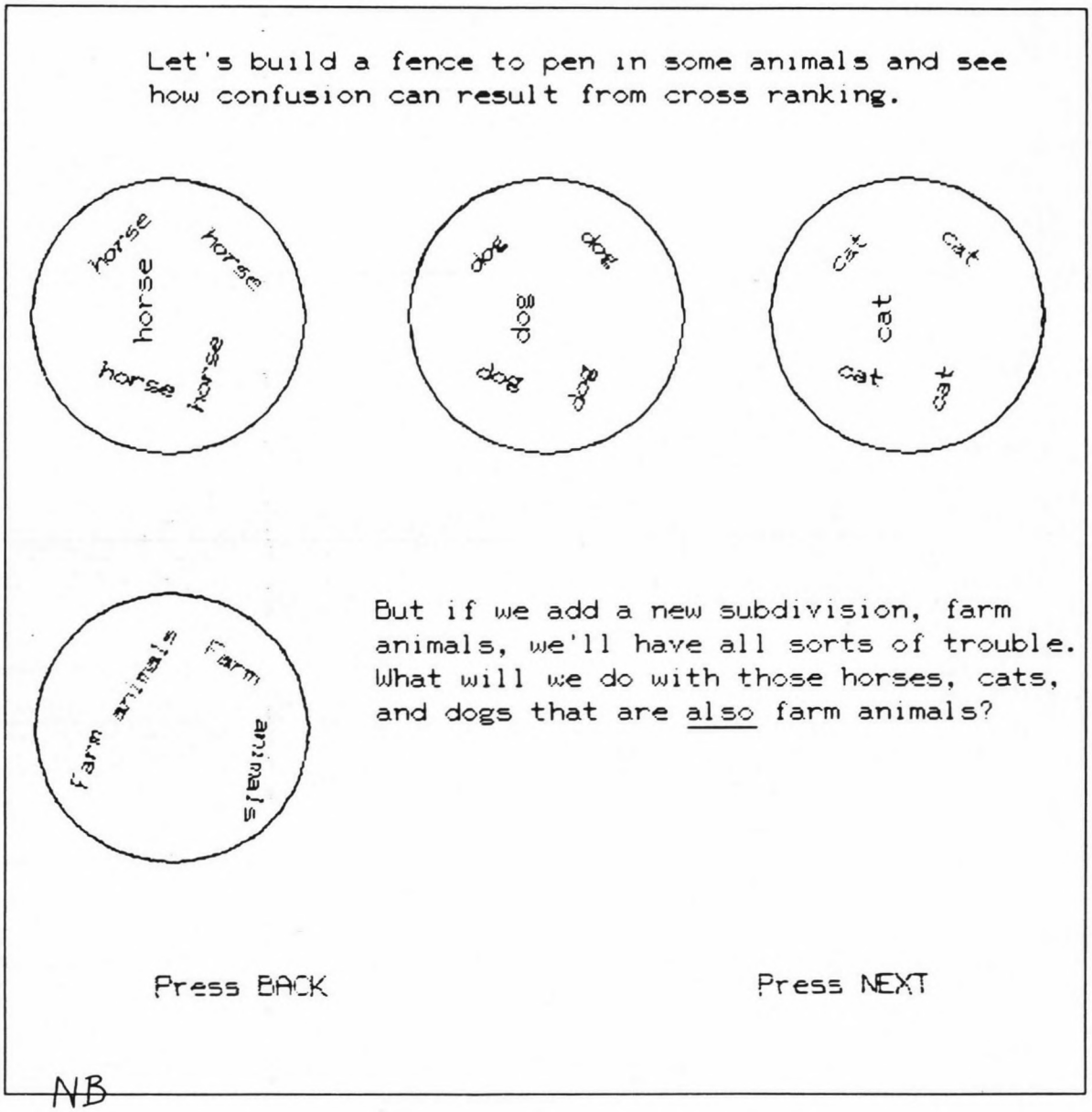


Special instructions:

(1) Bring up circles one at a time, separated by a pause. The student will press NEXT to receive each one.

(2) After all is on the display, erase this text. See next page for the revised display.

Demonstrate confusion
B



Additional instructions

1. Draw a box of two thicknesses around each of the items that designate the classification scheme in the three examples and in the problems.

   An example:

Literature into: English, American, French, Biography

2. Go through your activity and underline the pieces of text that are underlined in the following display.

Purpose of lesson

In judging the usefulness of a classification scheme, there are two important factors for one to consider.

One is that the subdivision must be significant; that is, the distinctions are worth making. This judgment is usually made relative to a particular purpose.

This lesson deals primarily with the second important factor: the subdivision must be made in terms of a single basis of division. Cross-ranking is to be avoided because it introduces confusion.

NB

8

Additional Judging Instructions

Previous units have discussed instructions that allow you to judge anticipated correct responses (-answer- and -answerc-) and incorrect responses (-wrong- and -wrongc-) that are treated as alphabetic characters. This unit presents instructions that allow you to store student responses in variables, record which anticipated responses are made, and judge numerical responses.

INSTRUCTIONAL OBJECTIVES

After completing this unit, you should be able to:

- Identify the effects of the -store-, -storea-, -showa-, and -ok- instructions
- Identify the effects of the -match- instruction
- Identify the effects of the -ansv-, -wrongv-, and -specs- *noops* instructions

LEARNING ACTIVITIES

_____ 8-A. Storing Student Responses. Text/Exercise/CAI: This activity presents the -store-, -storea-, -showa-, and -ok- instructions.

_____ 8-B. Matching Anticipated Responses. Text/Exercise/CAI: This activity presents the -match- instruction.

_____ 8-C. Judging Numerical Responses. Text/Exercise/CAI: This activity presents -ansv-, -wrongv-, and -specs- *noops*.

TESTING DIRECTIONS

Now sign on to Control Data PLATO terminal. If you are very familiar with any of the additional judging instructions mentioned on the previous page, you may wish to test out of some of the activities in this unit. If you are not familiar with these instructions, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.

8-A. STORING STUDENT RESPONSES



Identify the effects of the -store-, -storea-, -showa-, and -ok- instructions.

The -answer-, -answerc-, -wrong-, and -wrongc- instructions allow you to judge anticipated responses. In some applications, however, you will want the student to make a response that cannot possibly be anticipated. The student's name is one example. Another example is a number or arithmetic expression to be used in a calculation.

This activity discusses four instructions that are useful when the student's response cannot be anticipated: -store-, -ok-, storea-, and -showa-.

THE -store- AND -ok- INSTRUCTIONS

Suppose that you are writing a program that will convert degrees Fahrenheit to degrees Celsius. You want the student to specify degrees Fahrenheit; the program will calculate the corresponding degrees Celsius. This program requires the -store- instruction. The general form for -store- is:

```
store    VARIABLE
```

The tag is a variable in which the student's response is to be stored. The response must be a number or an arithmetic expression, for example, 2 x 14. The -store- instruction evaluates arithmetic expressions and stores the result, 28, in the variable. The -store- instruction is a judging instruction; that is, it is only executed in the judging state.

Only the first 1Ø characters of the student's response will be stored. If the student's response is 1234567891234, only 1234567891 will be stored.

The following example shows how -store- could be used in the program converting degrees Celsius to degrees Fahrenheit:

```
define   fahr     = v1
         cels     = v2
unit     convert
at       4Ø5
write    Degrees Fahrenheit?
arrow    where + 2
```

```
store    fahr
  .
  .
endarrow
  .
  .
```

These instructions ask the student to input a number signifying a temperature in degrees Fahrenheit. The number is then stored in the variable *fahr*. This variable could then be used in a -calc- instruction to convert degrees Fahrenheit to degrees Fahrenheit to degrees Celsius

```
calc     cels←(fahr-32)×(5/9)
```

and the result could be displayed with a -write- instruction, for example:

```
write    Degrees Celsius = ‹s,cels›
```

If the two instructions were added following the -endarrow-, however, they wouldn't be executed. The reason is that -store- terminates judging only when the response is not an appropriate arithmetic expression, when the response contains alphabetic characters (-store- expects numbers, not letters) or when the response is an arithmetic expression that cannot be evaluated because of missing arithmetic operations or numbers. If -store- terminates judging because the response is inappropriate, it makes a *no* judgment; when NEXT is pressed, the student's response is cleared and the PLATO system waits for a new response. If the response is an appropriate arithmetic expression, -store- does not terminate judging and the PLATO system continues to execute instructions in judging state. If the system doesn't encounter an instruction that causes a judgment and a switch to regular state, a *no* judgment will be made when the -endarrow- is encountered. The student's response will therefore always be judged *no* and instructions following the -endarrow- will never be executed. Something following the -store- instruction is needed to cause a switch back to regular state with an *ok* judgment.

What could be used? In unit 2, the -no- instruction is presented. This instruction terminates judging, but it also makes a *no* judgment. An appropriate instruction in this case is -ok-. Like -no-, -ok- is a judging instruction that terminates judging, but it makes the *ok* judgment. If -ok- (it has no tag) followed the -store- instruction, almost all student responses would be judged *ok*, judging would be terminated, and the -calc- and -write- instructions would be executed. The complete program for converting degrees Fahrenheit to degrees Celsius is:

```
define   fahr     = v1
         cels     = v2
unit     convert
at       405
```

```
write   Degrees Fahrenheit?
arrow   where + 2
store   fahr
ok
endarrow
calc    cels←(fahr-32)×(5/9)
at      605
write   Degrees Celsius = (s,cels)
```

Even though the -ok- instruction has been included, some student responses would not be judged *ok*. In some cases, the -store- instruction may terminate judging with a *no* judgment before the *ok* is executed. This happens when the response is *not* an arithmetic expression that can be evaluated. One example is when the student's response contains alphabetic characters; -store- expects numbers, not letters or words.

Another example is when the response is an incomplete arithmetic expression. If the student's response is an arithmetic expression, for example, 2 × 6, the -store- instruction evaluates the expression and stores the result, 12, in the variable. If the expression cannot be evaluated, if, for example, the incomplete expression 2 × is entered, -store- makes a *no* judgment.

If -store- makes a *no* judgment on a response, the student must press NEXT to clear the response and enter a new response. The -ok- instruction and following regular instructions are not executed until -store- accepts the response.

THE -storea- AND -showa- INSTRUCTIONS

The -store- instruction is used to store numbers. To store alphabetic characters or numbers that can be treated as alphabetic characters (numbers that will not be used in calculations), you need the -storea- instruction. You also need a different variable format when defining variables to be used with the -storea- instruction. Your variables must be preceded by an *n* rather than a *v* (*n1*, for example). You will learn more about the *n* variable format if you take PLATO Author Language: Part II. The general form of -storea- is:

```
storea  VARIABLE,NUMBER OF CHARACTERS
```

The tag contains the name of the variable in which the student's response is to be stored and the number of characters that can be stored. For example, the instruction

```
storea  name,15
```

allows up to 15 characters, including spaces, to be stored. Capital letters count as two characters: *california* counts as 1Ø characters; *California* counts as 11. If you wish to store 1Ø characters or less, the number of characters may be left unspecified; if the number of characters is not specified, the PLATO system will store up to 1Ø characters.

If you specify a number of characters greater than 1Ø, you must be careful about which variables you use. Each variable holds a 1Ø-character computer word (computer words are discussed in unit 1). Because no more than 1Ø characters can be stored in a variable, characters past the first 1Ø will be stored in the *next* variable. Suppose that you have the following instructions:

```
define  name     = n1
unit    example
at      4Ø5
write   What is your name?
arrow   where+2
storea  name,15
ok
endarrow
   .
   .
   .
```

If the student's response is *Warren Wonka,* the first 1Ø characters (*Warren W*) will be stored in the variable *name* (n1). The last 4 characters (*onka*) will be stored in the next student variable (n2). There is no problem here unless you try to use n2 for something else; storing a new value in a variable will destroy what was stored previously. If you specify a number of characters greater than 1Ø, be sure to reserve enough variables following the variable specified in the tag.

Remember that each variable can store 1Ø characters; thus,

```
storea  name
storea  name,8
```

or

```
storea  name,1Ø
```

will require one variable and

```
storea  name,12

storea  name,15

storea  name,2Ø
```

will require two consecutive variables. Do not use a variable that may be needed by a -storea- to store other information if there is any danger that the information will be overwritten by the -storea-, or vice versa.

When -ok- follows -storea-, *all* student responses are judged *ok*. Unlike -store-, -storea- does not evaluate the student's response (even numbers are treated as alphabetic characters) and, therefore, never ends judging. The -ok- instruction is always executed when it follows -storea-.

In order to display the contents of the variable *name*, you need a new instruction: -showa-. The -showa- instruction is like the -show- instruction discussed in unit 6, except that it is used to show alphabetic rather than numeric information. The general form for -showa- is:

```
showa    VARIABLE,NUMBER OF CHARACTERS
```

The tag specifies the variable that stores the value to be shown and the number of characters to be shown. If the number of characters is not specified, the PLATO system will show up to 1Ø characters. When you define a variable to be used with a -showa- instruction, you should, again, precede your variable with the letter *n* rather than the letter *v*.

The following instructions use -showa- to display a student's name:

```
define   name     = n1
unit     example
at       4Ø5
write    What is your name?
arrow    where+2
storea   name
ok
at       6Ø5
write    Hi␣
showa    name
```

The program first asks for the student's name. If the student enters *George*, the program writes *Hi George*. However, if the student enters *Mary Katherine*, the program writes *Hi Mary Kat*. If the response is likely to be longer than 1Ø characters, you must specify the number of characters to be shown,

```
storea   name,2Ø

showa    name,2Ø
```

and two consecutive variables must be reserved.

Note the space bar symbol following the letter *i* in the -write-. (The symbol indicates where the space bar is pressed.) Like -show-, -showa- doesn't add spaces.

The -showa- instruction can be embedded in a -write- instruction. The instruction

```
write   Hi ‹a,name›
```

is equivalent to the last -write- and the -showa- in the preceding example. The letter *a* is used to indicate an embedded -showa-. Remember that the letter *s* is used to indicate an embedded -show-.

If you have specified a number of characters other than 1Ø,

```
storea  name,8
```

or or

```
storea  name,15
```

the embedded format for displaying these in a -write- instruction is:

```
‹a,name,8›
```

or

```
‹a,name,15›
```

CHECK YOUR UNDERSTANDING

Section 1

Directions: Questions 1 through 8 refer to the following instructions. Select *all* the correct answers for each question.

```
define  name    = n1
        age     = v2
unit    example
at      4Ø5
write   What is your name?
arrow   where+2
storea  name
ok
endarrow
at      6Ø5
write   How old are you (in years)?
arrow   where+2
store   age
ok
```

```
endarrow
at        805
write     Well,⌧
showa     name
write     , I'm younger than⌧
show      age
```

_____ 1. If a student enters *Sandy Carson* for a name, what is stored in the variable *name*?

a. Sandy Carson
b. Sandy Cars
c. Sandy Ca
d. Sandy

_____ 2. Suppose that you changed the -storea- so that up to 15 characters would be stored. Which of the following -define- instructions could be used?

a. `age = v2`
b. `age = v3`
c. `age = v4`
d. `age = v5`

_____ 3. Which instruction could be used to replace the last four instructions in the example?

a. `write Well, (s,name), I'm younger than (s,age)`
b. `write Well, (a,name), I'm younger than (a,age)`
c. `write Well, (s,name), I'm younger than (a,age)`
d. `write Well, (a,name), I'm younger than (s,age)`

_____ 4. Which instruction would replace the -showa- if the -storea- were replaced by:

```
storea  name,15
```

a. `show  name,15` name,15
b. `showa  name,15`
c. `show  name`
d. The -showa- should not be replaced.

_____ 5. What would happen if the second -ok- instruction were deleted?

a. No instruction following the -store- would be executed.
b. Only the -show- instruction would not be executed.
c. Both the -show- and showa- instructions would not be executed.
d. Only the last two -write- instructions would not be executed.

_____ 6. What would happen if the expression 2 x 14 were entered at the second arrow?

a. The response would be judged *no*.
b. The value 28 would be stored in variable *age*.
c. The expression 2 x 14 would be stored in variable *age*.
d. The value 2 would be stored in *age*.

_____ 7. What would happen if the student's response at the second arrow were *twenty*?

a. Nothing; twenty would be stored in variable *age*.
b. The -store- instructions would make a *no* judgment.
c. 20 would be stored in variable *age*.
d. Twenty would be stored in variable *name*.

_____ 8. What would happen if the student's response at the first arrow were *3*?

a. Nothing; 3 would be stored in variable *name*.
b. The -storea- instruction would make a *no* judgment.
c. Three would be stored in variable *name*.
d. 3 would be stored in variable *age*.

Answers

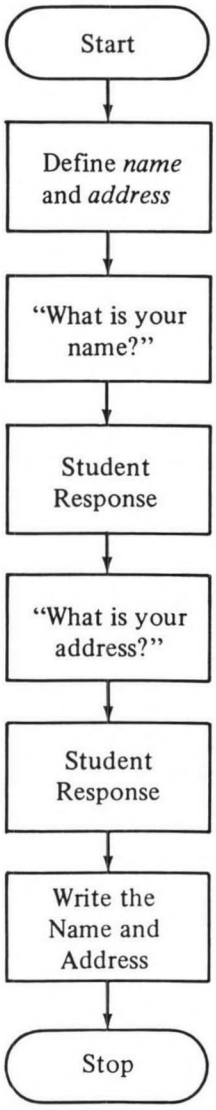
1. c. Only the first 1Ø characters, including spaces, will be stored. Capital letters count as 2 characters.
2. b, c,d. Because some of the characters stored in *name* will be stored in v2, v2 must be reserved.
3. d. An embedded -show- must be used to show variable *age*. An embedded -showa- must be used to show the variable *name*.
4. b. The instruction would have to specify 15 characters.
5. a. The student's response would always be judged *no* and instructions following the -store- would never be executed.
6. b. The expression would be evaluated and the result would be stored in the variable *age*.
7. b. -store- expects a number, not alphabetic characters.
8. a. -storea- treats all student responses as alphabetic characters.

Section 2

Directions: Write and execute a program that asks the student to enter a name and address and then writes the name and address entered. The program should be able to store and show the following:

Joe Smith
14685 North Lexington Avenue

Use -storea- and -showa-. Structure your program according to the following flowchart. One possible solution is given in the answer key. Be sure that you use the *n* variable format when defining variables to be used with -storea- and -showa-. This is true for both the name and address variables; in an address, the numbers are not used for calculations, so they should be treated as alphabetic characters.

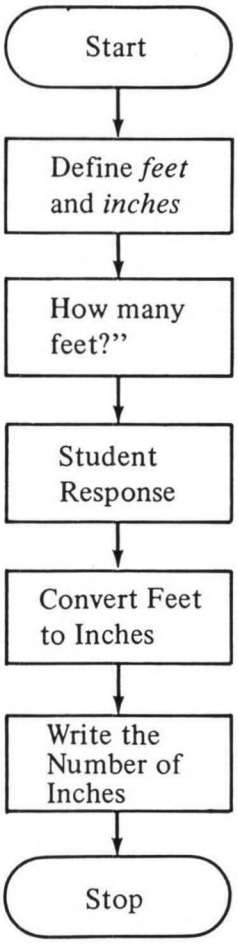


Answer Key

```
define  name     = n1
        address  = n3
unit    example
at      410
write   What is your name?
arrow   610
storea  name,20
ok
endarrow
at      810
write   What is your address?
arrow   1010
storea  address,40
ok
endarrow
at      1610
write   Name: (a,name,20)
        Address: (a,address,40)
```

Section 3

Directions: Write and execute a program that asks the student to enter a number representing feet, converts feet to inches, and writes the number of inches. Use -store- and -show-. Structure your program according to the following flowchart. Two possible solutions are given in the answer key. One does the calculation of feet to inches with the embedded -show-.



Answer Key

Solution 1:

```
define  feet     = v1
        inches   = v2
unit    example
at      410
write   How many feet?
arrow   610
store   feet
ok
endarrow
calc    inches←feet×12
at      1010
write   That's (s,inches) inches.
```

Solution 2:

```
define  feet     = v1
unit    example
at      410
write   How many feet?
arrow   610
store   feet
ok
endarrow
at      1010
write   That's (s,feet×12) inches.
```

8-B. MATCHING ANTICIPATED RESPONSES



Identify the effects of the -match- instruction.

The -answer-, -answerc-, -wrong-, and -wrongc- instructions allow you to judge anticipated responses. The -store- and -storea- instructions allow you to store responses. What if there are several anticipated responses and you want to judge each and be able to identify which response was made? For example, suppose that you are writing a lesson on how to interview job applicants. There are three applicants, and you want to allow the student to examine their application forms. The student is to indicate which forms are to be examined by entering the applicants' names. The student's response should be judged *ok* if one of the three names is entered. An indicator of the response made should be recorded so that a later instruction can branch to the appropriate forms. You could code each applicant's name in successive -answer- instructions and calculate a variable to represent the answer:

```
answer  karen
calc    person=0
answer  dennis
calc    person=1
answer  sandy
calc    person=2
```

There is a much more efficient way to solve this problem, however, using the -match- instruction.

THE -match- INSTRUCTION

The -match- instruction judges the student's response against a set of alternative responses. If the student's response matches one of the alternatives, a number corresponding to the identity of the match is stored. Unlike -store- and -storea-, execution of -match- always terminates judging. The general form of the -match- instruction is:

```
match   VARIABLE,ALTERNATIVE 1,ALTERNATIVE 2,
        ALTERNATIVE 3, ...
```

The alternatives given in the tag are alternative student responses. The -match- instruction judges the student's response *ok* if it matches one of the alternatives given. A numerical value is stored in the variable given in the tag depending on the alternative matched. If the match is with alternative 1, the value Ø is stored. A match with alternative 2 stores a 1, a match with alternative 3 stores a 2, and so on. If there is no match, -match- judges the student's response *no* and stores -1 in the variable.

You may have noticed the similarity of the format of -match- to -keytype-. Like -keytype-, -match- is not a conditional instruction. The variable in the tag does not indicate which listed alternative is to be performed; alternative student responses are listed, not alternative actions. The only possible action to be performed is to assign a value to the variable. The student's response determines which value is to be assigned to the variable. With -keytype-, the student's response is one key press; with -match-, the student's response can be a word or number with several key presses. In both cases, the value –1 is assigned to the variable if the student's response doesn't match one of the listed alternatives.

Here, the similarity ends; -match- is a judging instruction and is only executed in judging state. Keytype is executed only in the regular state.

The following instructions use -match- to solve the job applicants problem:

```
define  person  = v1
unit    hire
at      405
write   Which application forms would you
        like to examine?
arrow   605
specs   bumpshift
match   person,karen,dennis,sandy
writec  person#Choose Karen, Dennis, or Sandy.##
endarrow
jump    person,x,karen,dennis,sandy
unit    karen
  .
  .
unit    dennis
  .
  .
unit    sandy
  .
  .
```

If the student enters *karen*, the variable *person* is set to ∅, the -writec- performs the null alternative, and the -jump- instruction causes an immediate branch to *unit karen*. If the student enters *dennis, person* is set to 1 and the branch is to *unit dennis*. If the response is *sandy, person* is set to 2 and the branch is to *unit sandy*. If any other response is made, it is judged *no, person* is set to –1, and the program performs the first alternative of the -writec- instruction. Because no match is found and the response is judged *no*, the student is required to erase the response and try again.

The -match- instruction allows extraneous words in the student's response. For example, the response *I'll try Karen* would be judged *ok*. The reason for this is that -match- searches the student's response from left to right for any of the alternatives specified. If the first word in the response doesn't match any alternative, the second word is examined. If the second word doesn't match an alternative, the third word is examined. The search continues until a match is found or the last word has been examined. A *no* judgment is made when the last word in the response has been examined and does not match an alternative. For example, if the student's response is *How about Sandy or Dennis, How* would be examined first, then *about*, and finally *Sandy. Sandy* would match an alternative, and the variable *person* would be set at 2. The words *or* and *Dennis* would not be examined.

What if the alternatives have synonyms? For example, students may enter *Karen* or *Karen's*. Both should be considered the same response. Synonyms are enclosed in parentheses. The instruction

```
match   person,(karen,karen7s),(dennis,dennis7s),
        (sandy,sandy7s)
```

would set the variable *person* to ∅ if the response is *Karen* or *karen's*, to 1 if the response is *dennis* or *dennis's*, and to 2 if the response is *sandy* or *sandy's*. Notice that the -specs- bumpshift requires that the number 7 be used in the tag instead of an apostrophe.

Alternative responses can be phrases. The words in a single phrase must be connected by asterisks. For example, the instruction

```
match   person,karen*brown,dennis,sandy
```

would match the response *karen brown*.

Words in a phrase should never be separated by a space. A space is a delimiter, just like a comma. For example,

```
match   person,karen brown,dennis,sandy
```

is equivalent to

```
match   person,karen,brown,dennis,sandy
```

CHECK YOUR UNDERSTANDING

Section 1

Directions: Questions 1 through 6 refer to the following instructions. Select the correct answer for each question.

```
define  topic    = v1
        choice   = v2
unit    program
at      405
write   Which topic would you like to study?
          Judging
          Branching
          Calculations
arrow   905
specs   bumpshift
match   topic,judging,branching,calculations
  .
  .
  .
```

_____ 1. What is the value stored in variable *topic* if the student's response is *judging*?

a. -1
b. Ø
c. 1
d. 2

_____ 2. What is the value stored in variable *topic* if the student's response is *Both judging and branching*?

a. -1
b. Ø
c. 1
d. 2

_____ 3. If the instruction

```
storea  choice
```

is placed *after* the -match-, what is stored in variable *choice* if the student's response is *branching*?

a. branching
b. 1
c. one
d. Nothing. -Storea- would not be executed because -match- terminated judging.

_____ 4. If the instruction

```
storea  choice
```

is placed *before* the -match-, what is stored in variable *choice* if the student's response is *branching*?

a. branching
b. 1
c. one
d. Nothing. -Storea- would not be executed because -match- terminated judging.

_____ 5. If the -match- instruction is replaced with

```
match   topic,(judging,judge),branching,calculations
```

and the student's response is *judge*, what value would be stored in the variable *topic*?

a. -1
b. ∅
c. 1
d. 2

_____ 6. If you wanted to match *judging instructions* instead of *judge*, which instruction should be used?

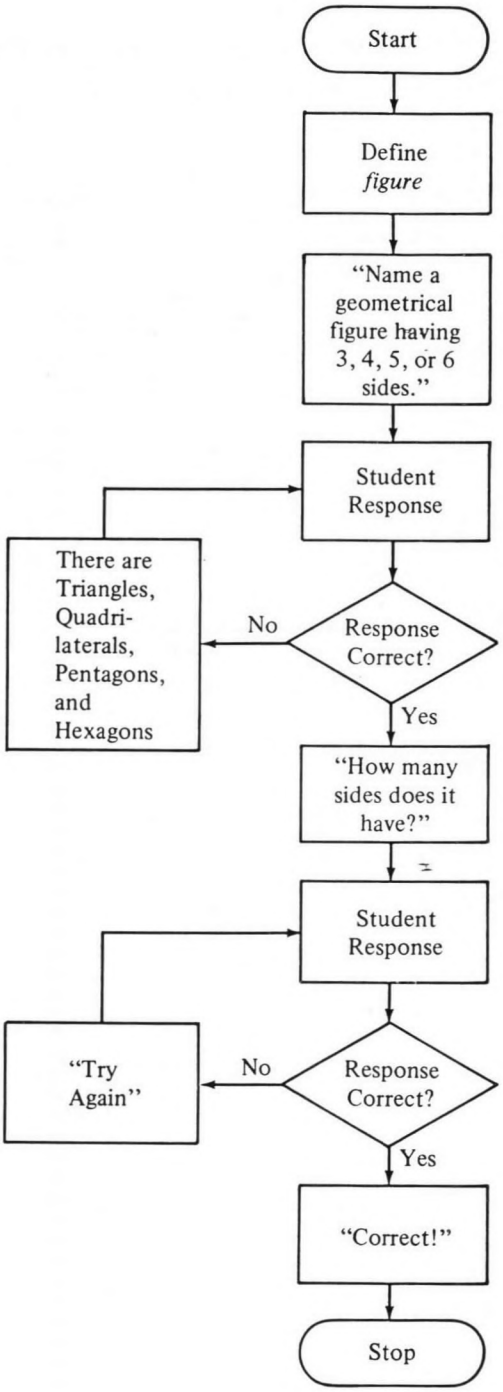
a. `match  topic,(judging instructions),branching,calculations`
b. `match  topic,(judging,instructions),branching,calculations`
c. `match  topic,judging instructions,branching,calculations`
d. `match  topic,judging*instructions,branching,calculations`

Answers

1. b. If the match is with the first alternative, the value ∅ is assigned to the variable.
2. b. The -match- instruction searches the student's response from left to right for any of the alternatives specified.
3. d. Because -match- terminates judging, the -storea- would not be executed.
4. a. Because the judging state has not been terminated, the -storea- will be executed.
5. b. A match with any of the alternatives within parentheses will cause the same value to be assigned.
6. d. The words in a phrase must be connected by an asterisk.

Section 2

Directions: Write and execute a program that asks the student to name a two-dimensional geometrical figure, asks the student to enter the number of sides the figure has, and provides appropriate feedback. The student should name a figure with 3, 4, 5, or 6 sides. Use -match-. Structure your program according to the following flowchart. One possible solution is given in the answer key.



Answer Key

```
define  figure  = v1
unit    example
at      41Ø
write   Name a two-dimensional geometrical figure
        having 3,4,5, or 6 sides.
arrow   61Ø
specs   bumpshift,okspell
match   figure,triangle,(quadrilateral,square,rectangle,rhom
        bus,parallelogram,trapezoid),
        pentagon,hexagon
writec  figure*There are triangles, quadrilaterals,
        pentagons, and hexagons.**
endarrow
at      81Ø
write   How many sides does it have?
arrow   1Ø1Ø
specs   bumpshift
answerc figure**3,three*4,four*5,five*6,six
write   Correct!
no
write   Try again.
endarrow
```

8-C. JUDGING NUMERICAL RESPONSES



Identify the effects of the -ansv-, -wrongv-, and -specs- *noops* instructions.

Suppose that you are writing an arithmetic drill. How do you judge the student's responses? You could use -answer- and -wrong-. For example:

```
unit     drill
at       405
write    What is 2 times 3?
arrow    605
answer   (6,six)
write    Right
wrong    (5,five)
write    No, you added.  Try again.
endarrow
```

This is alright for one problem, but if you wanted to present many problems, you would have a very long program. A better solution would be to use variables and set values with a random number generator. For example:

```
define   this     = v1
         that     = v2
unit     drill
randu    this,10
randu    that,10
at       405
write    What is ‹s,this› times ‹s,that›?
arrow    605
   .
   .
```

So far, so good. But how do you judge student responses? The -answer- and -wrong- instructions can't be used; they require specific answers, and the answers to these problems depend on the numbers generated. The -answerc- and -wrongc- instructions could be used but, like -answer- and -wrong-, these instructions treat responses as alphabetic characters and mark up incorrect answers in ways that will be inappropriate for numerical responses. (Answer markups are discussed in unit 2.)

This activity discusses two instructions that can be used in this situation: -ansv- and -wrongv-. A new -specs- tag, *noops*, is also discussed.

THE -ansv- AND -wrongv- INSTRUCTIONS

The -ansv- and -wrongv- instructions judge numerical responses. The response can be either a number or an arithmetic expression. The general form for -ansv- and -wrongv- is:

```
ansv     ANSWER,TOLERANCE
wrongv   ANSWER,TOLERANCE
```

The tag contains the answer against which the student's response is to be judged and a tolerance defining the accuracy of the judgment. The student's response will match the answer given in the tag if it is equal to the answer *plus or minus* the tolerance. For example, the instruction

```
ansv     6,1
```

judges the student's response *ok* if the response is equal to 6 plus or minus 1, or any response from 5 to 7. Tolerance can also be specified as a percentage. For example,

```
ansv     6,1%
```

judges the student's response *ok* if the response is equal to 6 plus or minus 1 percent of 6 (.06), or any response from 5.94 to 6.06. If the tolerance is not specified, it is zero; the student's response must equal the answer given in the tag to be judged *ok*. The answer need not be a number. It can be a variable, an arithmetic expression, or an arithmetic expression containing variables. For example:

```
ansv     product,10%
ansv     2*product
ansv     this*that
ansv     this+that
```

If the answer is given as an arithmetic expression, the expression is evaluated before the student's response is judged. For example, if variable *this* is set to 2 and *that* is set to 3, the instruction

```
ansv     this*that
```

is equivalent to:

```
ansv     6
```

Both instructions judge the response *6* as *ok*.

How can -ansv- and -wrongv- be used in the arithmetic drill? The following program uses -ansv- and -wrongv- to judge student responses. The -store- and -doto- instructions are also used to build a loop.

```
define  number  = v1
        this    = v2
        that    = v3
        count   = v4
unit    drill
  .
  .
at      2410
write   How many problems do you want?
arrow   where+2
store   number
ansv    10,9
no
at      2610
write   Choose a number between 1 and 19.
endarrow
doto    1done,count≠1,number
at      2410
erase   40
randu   this,10
randu   that,10
at      2410
write   What is ‹s,this› times ‹s,that›?
arrow   where+2
ansv    this×that
wrongv  this+that
at      2615
write   You added.
no
at      2615
writec  ntries>3≠The answer is ‹s,this×that›.≠≠
endarrow
1done
```

The first -write- instruction asks the student to specify the number of problems to be presented. The -store- instruction terminates judging if the response is not an appropriate arithmetic expression; otherwise, it stores the number specified in the variable *number*. The first -ansv- judges the student's response (the specified number of problems) *ok* if the number is between 1 and 19. If the number is not between 1 and 19, the response is judged *no* and the student must enter another response. Note that there is no need for an -ok- instruction following the -store-

in this example because the -ansv- judges the response *ok* if it is in the specified range and causes a switch to regular state; -ok- is used only when there is no other instruction to judge the response and cause a switch to regular state. The -doto- presents the specified number of problems. Two random numbers are generated. One is stored in variable *this*; the other is stored in variable *that*. The second -write- instruction uses these variables to present the problem. The -ansv- and -wrongv- use the variables to define answers. If the student's response matches the answer in -ansv-, the response is judged *ok*. If the student's response matches the answer in -wrongv-, the response is judged *no* and *You added* is written. If the response fails to match either answer, it is judged *no*. After three unsuccessful attempts, the student is told the correct answer.

THE *noops* TAG

If the student enters an arithmetic expression, the PLATO system evaluates it before it looks for a match. For example, if the student enters 2 × 3, the response is evaluated and taken to be 6. Thus, if the problem is

What is 2 times 3?

the student could enter

2×3

and -ansv- would judge the response *ok*.

There is a -specs- tag that provides a solution to this problem: *noops*. The *noops* tag indicates that the student's response may not contain any arithmetic operators (*noops* is an abbreviation for "no operators"). If -specs- *noops* is inserted after the second arrow in the arithmetic drill program, any response containing an operator will be judged *no*.

CHECK YOUR UNDERSTANDING

Section 1

Directions: Questions 1 through 4 refer to the following instructions. Select the correct answer for each question.

```
calc     a←4
         b←3
at       2Ø1Ø
write    What is ‹s,a› + ‹s,b›?
arrow    where+2
ansv     a+b
wrongv   a×b
at       211Ø
write    You multiplied.  Try again.
wrongv   a+b,1
at       211Ø
write    You're off by 1.  Try again.
endarrow
   .
   .
```

_____ 1. What happens if the student's response is 3 + 4?

a. The response is judged *ok* and *Correct* is written.
b. The response is judged *no*.
c. The response is judged *no* and *You multiplied. Try again.* is written.
d. The response is judged *no* and *You're off by 1. Try again.* is written.

_____ 2. What happens if the student's response is seven?

a. The response is judged *ok* and *Correct* is written.
b. The response is judged *no*.
c. The response is judged *no* and *You multiplied. Try again.* is written.
d. The response is judged *no* and *You're off by 1. Try again.* is written.

_____ 3. What happens if the student's response is 6?

a. The response is judged *ok* and *Correct* is written.
b. The response is *no*.
c. The response is judged *no* and *You multiplied. Try again.* is written.
d. The response is judged *no* and *You're off by 1. Try again.* is written.

_____ 4. What happens if -specs- *noops* is inserted after the -arrow- and the student's response is 4 + 3?

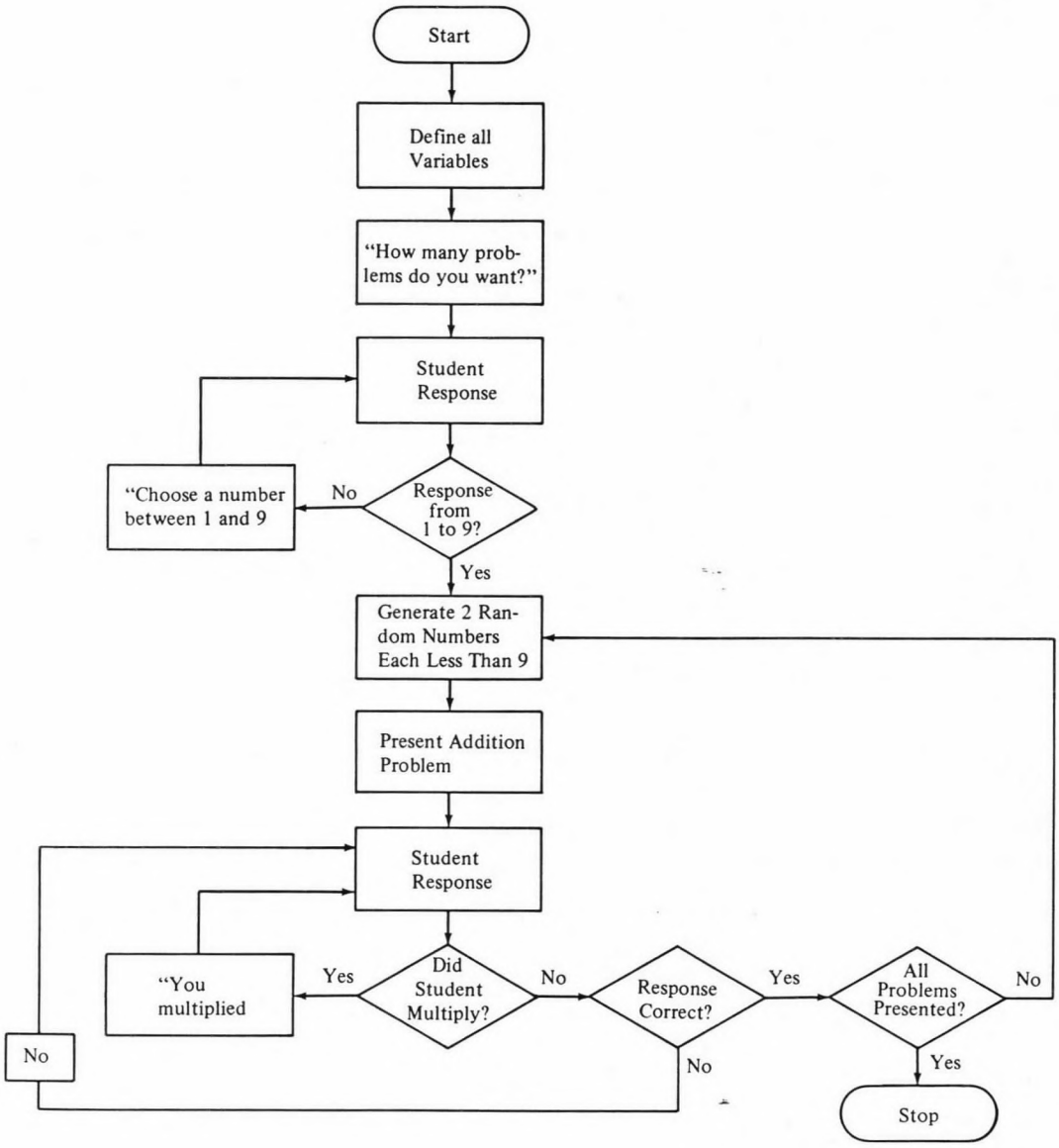
a. The response is judged *ok* and *Correct* is written.
b. The response is judged *no*.
c. The response is judged *no* and *You multiplied. Try again.* is written.
d. The response is judged *no* and *You're off by 1. Try again.* is written.

Answers

1. a. Because the student's response is treated as an arithmetic expression to be evaluated, the response 3+4 is taken to be 7.
2. b. No judging instruction tag is matched.
3. d. There will be a match at -wrongv- if the response is either 6 or 8.
4. b. -specs- *noops* indicates that the student's response may not contain any arithmetic operators.

Section 2

Directions: Write and execute an arithmetic drill that presents from one to nine addition problems. The student should specify the number of problems desired. The numbers in the problem should be generated randomly and should range from 1 to 9. The student must answer a question correctly before receiving the next question. If the student's response suggests that the numbers were multiplied, *You multiplied* should be presented as feedback. Use -ansv- and -wrongv-. Structure your program according to the following flowchart. One possible solution is given in the answer key.


Start
Define all Variables
"How many problems do you want?"
Student Response
"Choose a number between 1 and 9
No
Response from 1 to 9?
Yes
Generate 2 Random Numbers Each Less Than 9
Present Addition Problem
Student Response
"You multiplied
Yes
Did Student Multiply?
No
Response Correct?
Yes
All Problems Presented?
No
No
No
Yes
Stop

Answer Key

```
define  number  = v1
        this    = v2
        that    = v3
        count   = v4
unit    drill
at      1410
write   How many problems do you want?
arrow   1510
store   number
ansv    5,4
no
write   Choose a number between 1 and 9.
endarrow
doto    1done,count÷1,number
at      1410
erase   40,2
randu   this,9
randu   that,9
at      1410
write   What is (s,this) plus (s,tha
arrow   1510
specs   noops
ansv    this+that
wrongv  this*that
at      1610
write   You multiplied.
endarrow
1done
```