Fundamentals of PLATO® Programming

by

Celia R. Davis

ACKNOWLEDGEMENTS

I would like to thank Elaine Avner, Richard Blomme, William Golden, and Steven Williams for reading early versions of this book. Their thoughtful comments helped to clarify many ideas presented here. Those obfuscations which remain do so in spite of the efforts of my colleagues. I am grateful to Daniel Davis for advice on many sections of the book and for extensive assistance with chapter 5. I thank Wayne Wilson for the art work. And I am obliged to those PLATO authors who used drafts of the book and offered many useful suggestions.

Celia R. Davis
Urbana, Illinois
February, 1980

Table of Contents

Chapter 1: Resources for the PLATO Author

PLATO is a unique interactive computer system developed at the Computer-based Education Research Laboratory, University of Illinois and made available world-wide by Control Data Corporation. It can bring computer assisted instruction to hundreds of students at once. At the same time that students are working with the system, authors of computer-assisted instruction (CAI) materials can use the system to prepare and test new materials.

PLATO is a large central computer with many terminals connected to it. Each terminal consists of a keyset, a display panel, and the electronics necessary for communicating with the central computer. The keyset, by which the user communicates with the computer, is similar to a typewriter keyboard. Figure 1.1 shows a diagram of the keyset. The computer communicates to the user through a display panel capable of displaying text and graphics. This panel may be either a plasma panel (a black screen with orange display) or a cathode ray tube (a black and white television screen).

PLATO systems are used by colleges and universities, elementary schools, high schools, community colleges, medical colleges, government and business installations, and other institutions. Terminals are connected to the computer by telephone lines, and so may be located anywhere.

Educational materials have been written in more than 200 subjects ranging from accountancy to zoology. Examples include the biological, physical, and social sciences, medicine, law, art, English, and many foreign

languages. The level of materials ranges from elementary mathematics to college algebra and calculus, from beginning reading to college rhetoric.

The instructional advantages to using PLATO are immense. The rich graphic capabilities of PLATO permit the use of drawings, diagrams, and graphs as well as text to enhance presentations of ideas. In analysing and judging a student's typed responses, PLATO is not limited to true/false or multiple choice approaches; indeed, by following the author's instructions, PLATO interprets student responses expressed in a variety of ways, recognizing, for example, "car" and "vehicle" as synonomous in some contexts and different in others. As appropriate, "1/1Ø", "one tenth" and "a tenth" can be recognized as equivalent to "Ø.1". Perhaps more importantly, PLATO instructional materials can be responsive to the particular needs of individual students. The author can instruct PLATO to follow each student's progress through the materials, and to present to each student any appropriate supplemental material.

The advantages of the PLATO system for the author of computer-based lesson material are many. Portions of a lesson may be written and tested immediately without any waiting. Lessons may be quickly and frequently revised as the needs of students indicate. PLATO has "utility packages" which facilitate creating lesson displays and special characters or pictures. A large on-line reference manual is available at the press of a key. In addition, PLATO provides unparalled communications facilities. Authors can engage in typed conversations with other authors anywhere on the system, and can communicate with consultants who can advise and assist with any problems in using the system or writing lesson materials. Through electronic mail, authors can communicate privately. There are also informal discussion forums in which all authors and the PLATO staff may participate. Authors' suggestions for improvements to the system can be made in notes and are discussed by the entire PLATO community. Through such discussions, the PLATO system evolves in response to the needs of its users.

All this activity: students working, authors writing lesson materials, people engaging in discussions, can occur at once because PLATO is a time-sharing system. This means that the computer rapidly moves from one user to the next, doing whatever that user requires. Each user's turn or "time slice" lasts only a few thousandths of a second, but in that time the computer, because of its great speed, can receive the user's input, determine what action is required, and send output back to the user's terminal.

Standard PLATO Keyset

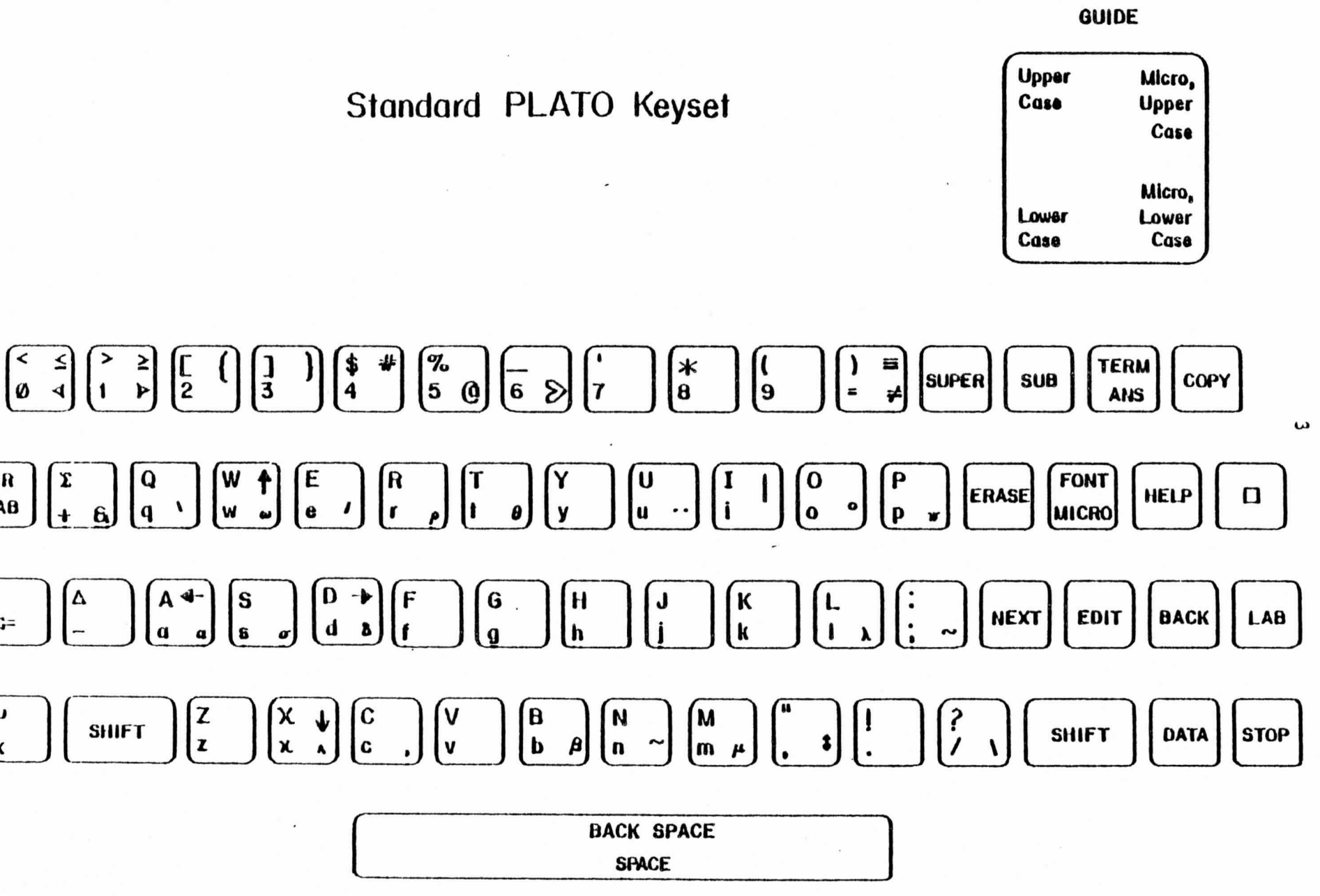


Figure 1.1

How the PLATO Community is Structured

PLATO users are organized into groups of people with common interests. A group might consist of all the authors at a PLATO site, or all the mathematics authors, or all students in a particular PLATO course, for example. One member of each group is the group director, who is responsible for creating PLATO records for members of the group and generally overseeing the activities of the group.

There are three types of PLATO users. Students can study the lessons assigned by their teachers and communicate with their teachers through online notes. Instructors can assign lessons to students and examine data showing the students' progress, use the communications facilities, and study lessons. Authors can create lesson material as well as assign lessons to students and examine student data, use the communications facilities, and study lessons. A fourth type of user record, called a multiple, is used for demonstrations of the PLATO system.

Three pieces of information uniquely identify each PLATO user: a PLATO name chosen by the user (usually the last name or first and last name), the name of the PLATO group to which the user belongs, and a password chosen by the user and known only to him.

Signing In To PLATO

Figures 1.2 through 1.4 show the signon displays where you type your name, group, and password. The character ≫, called an arrow, signifies that PLATO is waiting for your typed input. When you finish typing, press the key marked NEXT to show that your typing is completed.

After you sign in, you reach the Author Mode page. This is the author's home base, the display from which one accesses most authoring activity. Figures 1.5 and 1.6 show in flow chart form the features available from the Author Mode page. Pressing HELP from the Author Mode page shows a list of the features available. SHIFT-DATA gives a list of "Author Options". Five of these are of immediate interest: AIDS, the bulletin board, the catalog of lessons, notes, and personal notes.

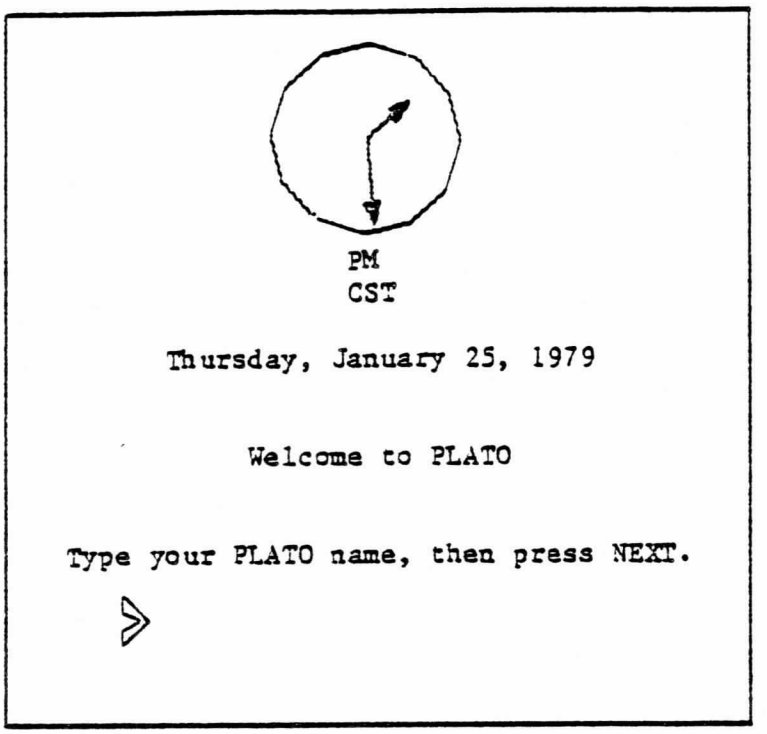


Figure 1.2
Name Page

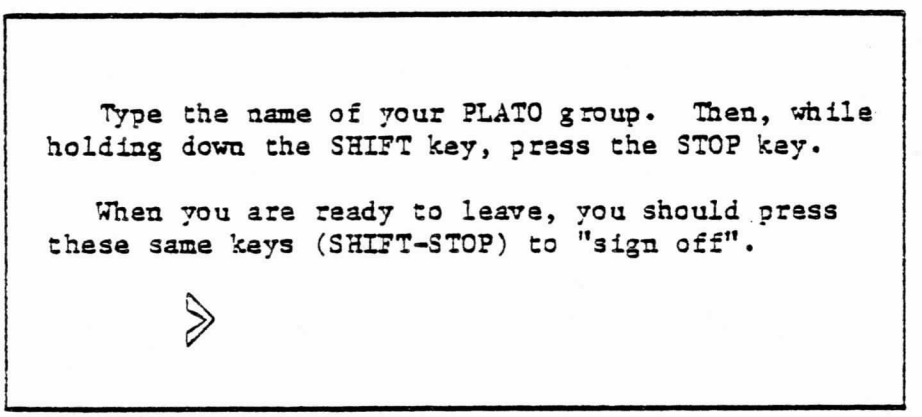


Figure 1.3
Group Page

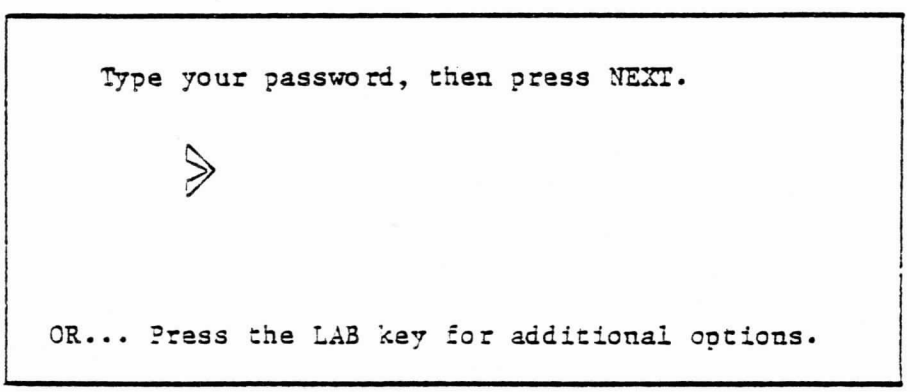


Figure 1.4
Password Page

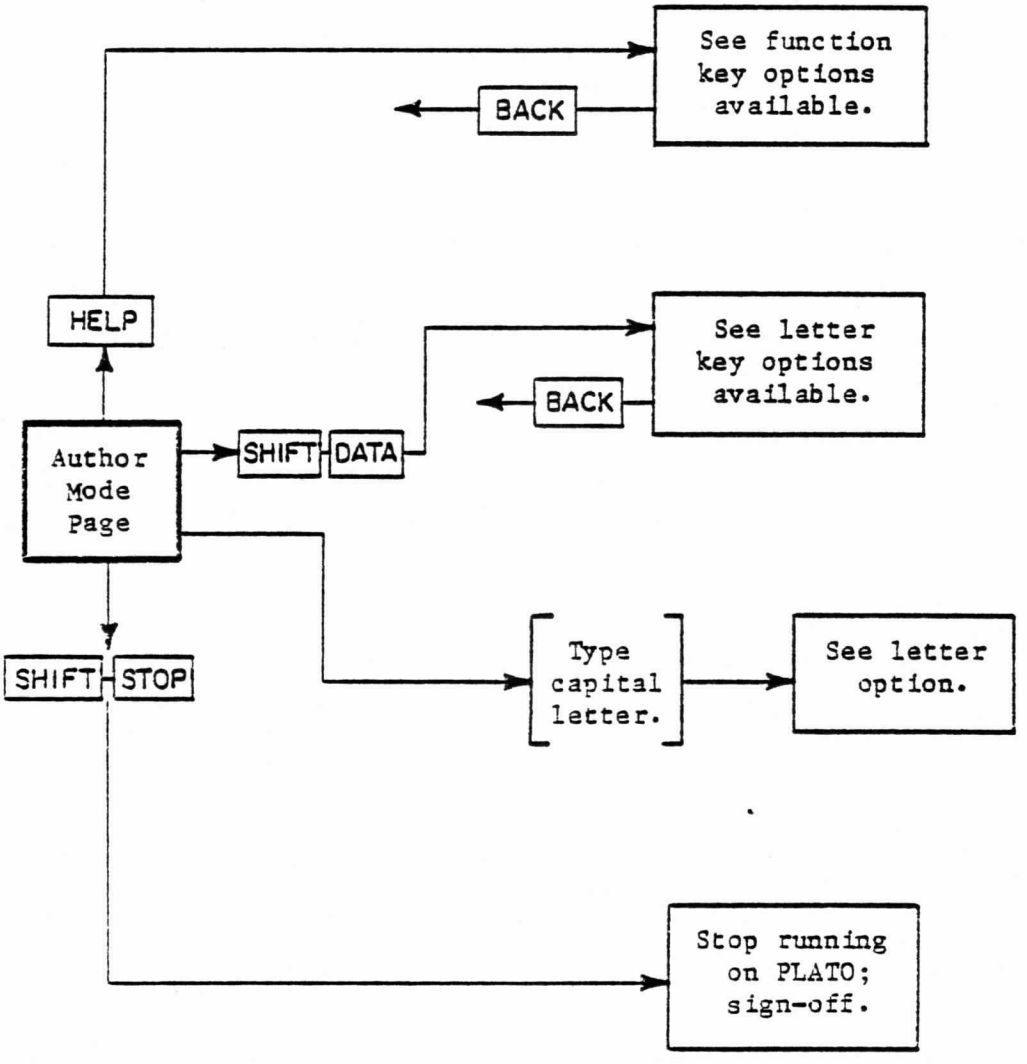


Figure 1.5
Some Features Available from Author Mode

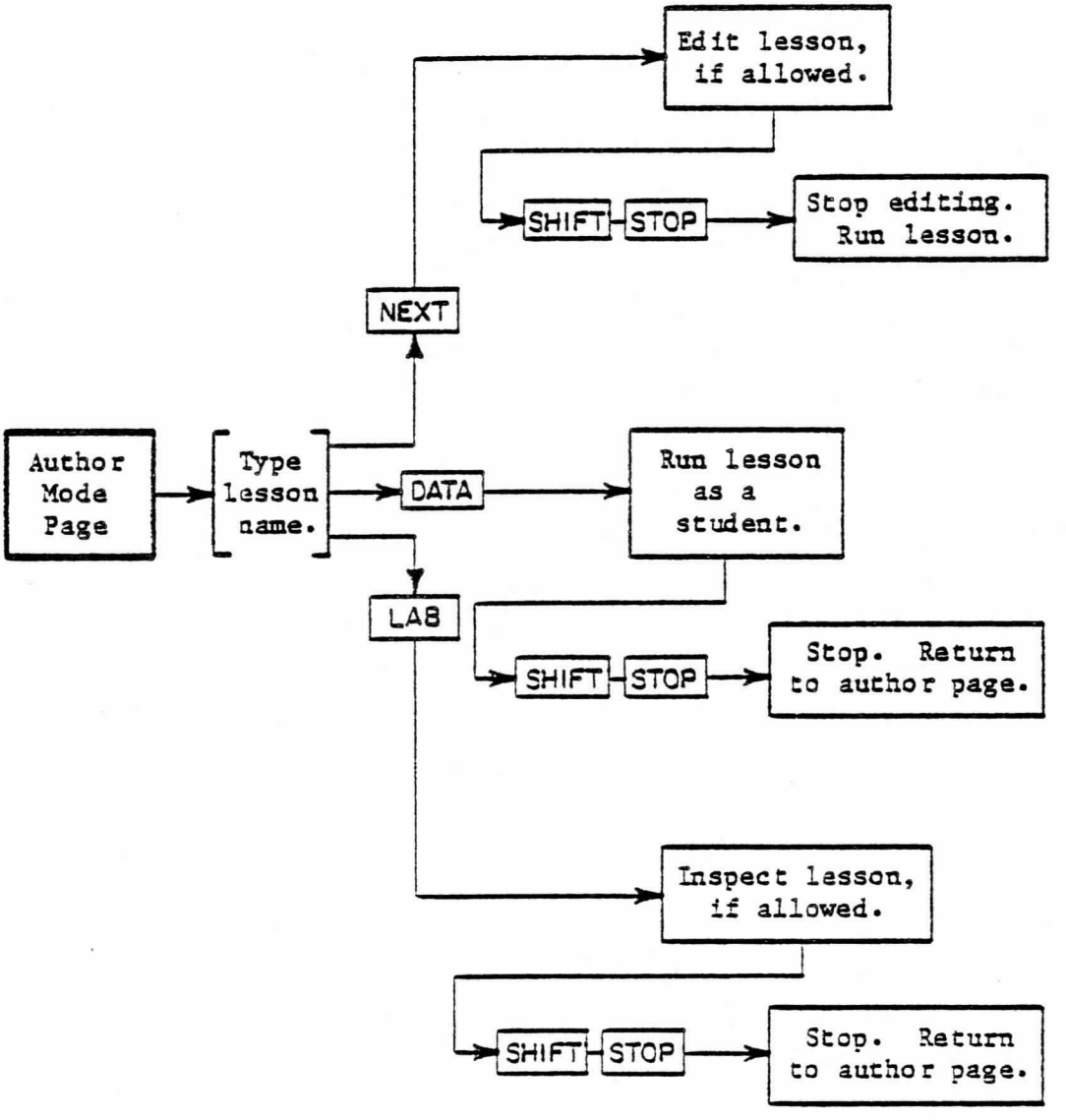


Figure 1.6
More Features Available from Author Mode

AIDS

The reference manual for the PLATO system and the TUTOR language, in which PLATO lesson materials are written, is available on-line rather than in book form. It is lesson AIDS. AIDS can be used for quick reference or for extensive information. The first several displays of AIDS comprise an index to overviews of general areas of TUTOR. The overviews present features of TUTOR in context with related features and give complete details. Figures 1.7, 1.8, and 1.9 show the title page and main index of AIDS. The new author's first selection in AIDS should be the section entitled "How to Use AIDS". As you use this introductory book you will often want to refer to AIDS for an expanded discussion of a particular topic. After finishing this book, you will use AIDS daily as you program, both for learning about an area of TUTOR new to you and for quick reference about details of TUTOR.

AIDS contains a wealth of information not only about the TUTOR language, but about all aspects of PLATO. The "Author Resources" section gives details about how to get consulting help, explanation of some hardware terms, names and phone numbers of PLATO personnel, names of helpful lessons, and so forth. AIDS has instructions about how to get a print of your lesson, how to collect data from students using your lessons, how to organize lessons into coherent modules which follow each other in an instructional plan, and much more. Also included are libraries of special characters and a library of general programming routines.

Bulletin Board

The Bulletin Board as announcements of general interest to the PLATO community. These announcements are made by PLATO personnel and concern recent improvements to the system, changes in the times of service, and so forth.

Catalog of Available PLATO Lessons

A catalog of PLATO lessons is available from the Author Mode display. Descriptions of student-tested lessons are given along with information about prerequisites, educational level, time for completion, etc. Provision is made for trying out the lesson.

AIDS

Elaine Avner, Darlene Chirolas
Celia Davis, Jim Ghesquiere,
Tina Gunsalus, Jim Kraatz,
and Judy Sherwood

PSO Author Group -- CERL
Univ of Illinois, Urbana

Press HELP if this is your first time in lesson AIDS

1879 features requested per day for the last 1 days

Figure 1.7
AIDS Title Page

page 1
Press a letter; or press NEXT for page 2 of 2

a Aids for new authors

b How to use AIDS

c Author Resources

d Alphabetical list of TUTOR commands

e Functional lists of TUTOR commands

f List of Indexes in AIDS

g Lists of System Defined Variables, Keynames, Functions, Logical & Bit Operators, -specs- Tags

h Making Displays

i Making Graphs & Charts

j Calculations and Variables

k Conditional Operations

l Sequencing

m Judging

n Execution of TUTOR

SHIFT-BACK always returns you to an index display.
HELP, DATA, BACK, SHIFT-NEXT are always available.

Figure 1.8
Main AIDS Index, page 1

Press a letter; or press NEXT for page 1 page 2 of 2

o The PLATO Computer

p Special Characters: ACCESS Characters, Linesets, FONT Characters (Character Sets), & MICRO Keys

q Student Data, Instructor Options, Routers

r Keynames, Keycodes, and Internal Codes

s Programming Errors and Condense, Lesson, & Execution Errors

t Library of Author Routines

u Microfiche and Photographing the Plasma Panel

v Systematic Lesson Design

w The Programmable Terminal (PPTs & TSTs)

x Changes to "Summary of TUTOR Commands"

SHIFT-BACK always returns you to an index display.
HELP, DATA, BACK, SHIFT-NEXT are <u>always</u> available.

Figure 1.9
Main AIDS Index, page 2

Notes

Lesson "notes" is the main avenue of communication among PLATO users. To use it, type "notes" or "N" on the Author Mode display. Figure 1.1Ø shows the index page of "notes". Lesson "notes" gives access to all notes files, including two files maintained by PLATO staff, System Announcements and Public Notes. "System Announcements" contains announcements of changes to TUTOR and changes in the way PLATO functions. New commands and new options for old commands are announced here. You should check System Announcements regularly, because changes might be made which affect your work. "Public Notes" is a forum for users concerning matters about PLATO of general interest, requests for help with programming problems, reports of possible malfunctioning of the system, etc. Authors should read Public Notes daily to remain well informed.

The Other Notes option gives access to all other notes, which may be on any topic and may be open to any users designated by the director of the file. Of particular interest to new authors is the notes file "psonotes", used for notes to members of the PLATO Services Organization. You may use this file to make comments about AIDS or any other PSO service or to request help. A member of PSO will respond to your request. File "notesfiles" contains a list of the names of other notes files.

Personal Notes

Personal Notes are private notes between individuals. You may write personal notes to any author or instructor whose group is prepared to receive personal notes. You may read personal notes addressed to you.

Interactive Communication

When the TERM key (notice that TERM is the shifted ANS key) is pressed, the message "What term?" and an arrow appear near the bottom of the screen. By typing the word "talk" at this arrow, you may converse with another author or instructor via the terminal.

To initiate a conversation, press the TERM key and type "talk". Then specify the PLATO name and the group of the person you wish to talk to. To accept a request for "talk", press TERM and type "talk". If you do not want to accept the call, press TERM and type "reject". If you try to "talk" to

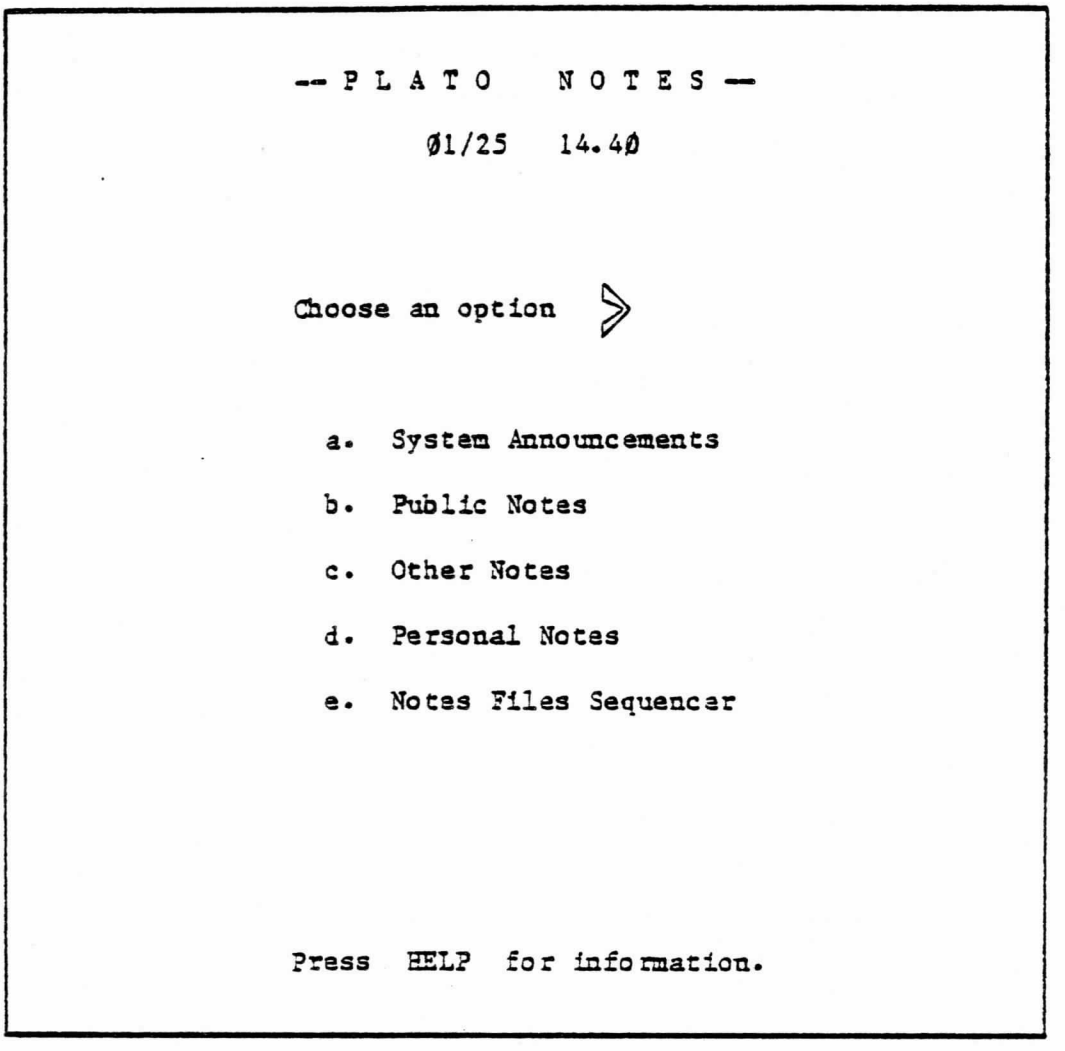


Figure 1.1Ø
PLATO Notes

someone and receive the message "xxx is busy, but has been notified of your call" it means he is talking to someone else. The message "You cannot talk to xxx" means that he has chosen not to receive "talk" calls. The message "rejected" means he has pressed TERM and typed "reject". During the conversation, each person sees an arrow near the bottom of the screen where his typed communication will appear. Both lines are shown to both people. A "talk" is ended when either person presses SHIFT-BACK.

For help with programming or using PLATO, press TERM and type "consult". All consultants on the system will be notified of your request, and one will respond to it as soon as possible. When the consultant answers your call, you will see a message on your screen saying that the consultant also sees your display. To talk to the consultant, press TERM. You need not type a special word at the TERM arrow. An arrow will appear at the bottom of the screen, and you can type there. To write a new line, press LAB to clear the line, then continue typing. During the consultation the consultant can see your display. When the consultation is completed, the consultant will break the connection.

If you find the answer to your question before the consultant answers, you can cancel the request: press TERM and type "consult" again. Then you may either reaffirm or cancel the request.

Some Other Resources

Locations of all terminals connected to your PLATO system can be seen in lesson "network". Lesson "authors" contains the names, addresses, and PLATO names of all authors who have chosen to have their names listed. A list of people currently using the system can be seen by pressing "U" from the Author Mode page.

Exercises

1. Ask your group director to show you how to sign on and sign off. Practice the sequence several times.

2. Familiarize yourself with the keyboard by doing one of the lessons designed to introduce the keyboard. Your group director or a PSO consultant can give you the names of several such lessons.

3. Look at several different lessons to become familiar with the range of styles used in PLATO lessons. Choose lessons suggested by your group director or make a selection from the catalog (see exercise 5).

4. Look at the Bulletin Board.

5. Examine the catalog of available lessons and learn how it is organized.

6. Read the section of AIDS entitled "How to Use AIDS".

7. Read today's Public Notes.

8. Write a personal note to yourself and another one to someone in your group or someone in the PSO group. (Names and groups of PSO members can be found in the Author Resources section of AIDS.)

Chapter 2: Main Units and Attached Units

A computer program is a set of precise instructions for carrying out a specific task. A computer can do only those tasks which it is programmed to do, neither more nor less. The complexity of the task any computer can perform depends to a large extent on the power of the programming language used. The TUTOR programming language, used on the PLATO system, is a very powerful language, having more than 300 commands; yet it is relatively easy to learn, and sophisticated instructional lessons can be written using only a small number of commands. TUTOR statements have two parts, a command and a tag. The command indicates a general instruction and the tag gives specific information for carrying out the general instruction. Here are two TUTOR statements:

```
at      1620
write   Let's begin.
```

They cause the computer to write the sentence "Let's begin." on the screen at the point 16 lines from the top and 20 spaces from the left edge of the screen.

The screen has 32 horizontal lines, each with 64 spaces. The upper left corner of the screen is position 101, the upper right corner is position 164, the lower left is position 3201, and the lower right is position 3264. The center of the screen is position 1632. Figure 2.1 shows these screen locations. A screen location, while written as a single number, actually

contains two pieces of information, the line and the space. The last two digits refer to the space. The first digit (in the case of lines 1-9) or the first two digits (in the case of lines 1Ø-32) indicate the line. Thus location 243 is 2 lines from the top and 43 spaces from the left, not 24 lines down and 3 spaces from the left. This latter position is location 24Ø3 (see figure 2.2).

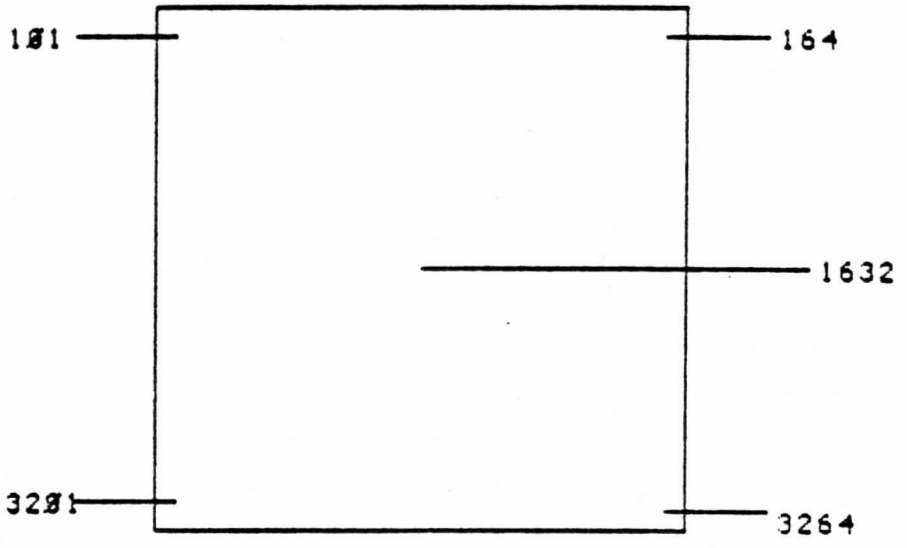


Figure 2.1 Locations on the Panel

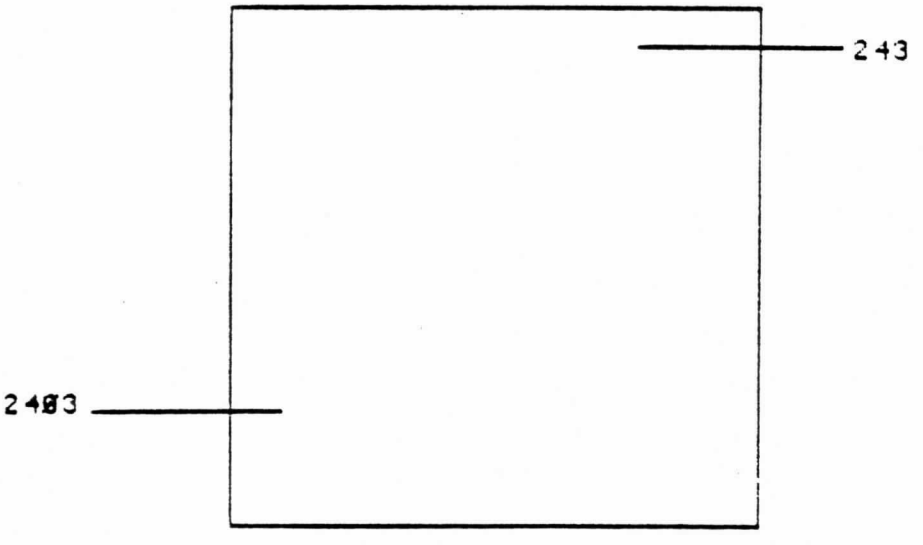


Figure 2.2 Some Screen Locations

PLATO lessons, or programs, are subdivided into units. Each unit contains a small group of instructions describing what is to be shown on the screen, whether the student is to respond, and if so what is to be done for every possible student response. The unit may also contain instructions indicating how the unit connects to other units in the program.

The concept of a TUTOR unit may be defined in several different ways. From a strict computer language point of view a unit is the set of instructions headed by the unit identifier. Because the PLATO system interacts with a student by presenting information to the student and reacting to his responses, we may think of a unit in educational terms: a sequence of presentation of information and reaction to student responses. From a practical standpoint it is sometimes helpful to think of a unit as a single screen display, although often a screen display consists of many units.

Shown below is some representative TUTOR code. (Code is the term used for a collection of program instructions.)

```
unit    title
next    topics
at      1623
write   Film History
box     1621;1536
*
unit    topics
next    adirect
at      1005
write   Topics covered in this lesson:
draw    1005;1035
at      1310
write   American Directors of the Thirties

        Influential Films of the Thirties

        Significant Contributions of European Directors
```

Each TUTOR unit begins with a -unit- command. (In this book, names of TUTOR commands are set off by hyphens. The hyphens are not used in actual TUTOR code.) The tag of the -unit- command is the name of the unit. Every unit in a lesson must have a unique name. It may be any name you like but may not be longer than 8 characters.

The -next- command tells which unit will be executed when the student completes the present unit and presses the NEXT key. The tag of the -next- command is the name of the next unit to be executed. In the example code, when the student completes unit "title" and presses NEXT, he proceeds to unit "topics". After completing unit "topics", he presses NEXT to proceed to unit "adirect". When a new main unit is entered, the screen display of the previous unit is erased.

The -at- command indicates a position on the screen for subsequent display, as for instance with a -write- command. The -write- command displays text on the screen. The tag of the -write- is the text to be shown. In unit "title" the words "Film History" are displayed at the location given by the preceding -at- command, that is, 16 lines from the top and 23 spaces from the left. The -box- command draws a box with one corner at the first screen location given in the tag and the other corner at the second location given in the tag. The -box- command in unit "title" draws a box around the words "Film History".

The -draw- command draws a line between the points specified in the tag. The -draw- in unit "topics" draws a line from location 1ØØ5 to location 1Ø35, underlining the -write- statement above it. The second -write- statement in unit "topics" illustrates several features of the -write- command: the tag may continue for several lines and blank lines may appear. Each subsequent line of text appears on succeeding lines when the lesson is used in student mode. The blank lines in the continued tag constitute double spacing.

The asterisk between the two units is not an instruction to the computer; it is a so-called comment symbol. Lines in TUTOR which begin with this symbol are ignored by the computer; they are for the author's convenience. They serve to increase readability of the program by separating sections of the lesson. Comments explaining the workings of the program may be written after the asterisk. Another comment symbol is $$, the double dollar sign. It may appear anywhere in the line. Any material on the same line after the $$ is taken as a comment; that is, it is not executed. The example units "title" and "topics" are shown below, revised to include programmer's comments.

```
unit    title
next    topics
at      1623   $$ 16 lines down, 23 spaces over
write   Film History
box     1621;1536
*
unit    topics
next    adirect    $$ Amer. directors of 3Ø's
at      1ØØ5
write   Topics covered in this lesson:
draw    1ØØ5;1Ø35
at      131Ø
write   American Directors of the Thirties

        Influential Films of the Thirties

        Significant Contributions of European Directors
** may add more topics after first semester
*
```

Good programmers use comments freely to document their lessons, thereby increasing the ease with which colleagues can understand the programming and eliminating delay when they themselves take up work on the program after an interval of several days or weeks.

When a student completes a unit and presses NEXT to proceed to the next unit, the screen is erased. The -next- command is used to specify which unit is to be done next; in the absence of a -next- command, the unit physically following the current one is the next unit. Units need not be placed in a lesson in sequential order since the order of presentation can be controlled by -next- commands. It is a good programming practice to use explicit -next- commands rather than to rely on the physical order of units, because you will occasionally find that you must move units from one physical location to another. If you know that the program's flow of control is explicitly stated in the code, you need not spend time adding -next- commands after reorganizing the physical location of units.

The -back- command can be used for review of previous units. The tag of the -back- command is the name of the unit to proceed to when the BACK key is pressed. The two units below illustrate the use of the BACK key and the -back- command. If the student presses BACK while in unit "irreg", he returns to unit "regular".

```
unit    regular
next    irreg
at      1412
write   The endings for regular -er French verbs are:

            e       ons
            es      ez
            e       ent
*
unit    irreg
next    irreg2
back    regular  $$ review permitted
at      121Ø
write   The verb "faire" is conjugated:

            je fais         nous faisons
            tu fais         vous faites
            il fait         ils font
at      282Ø
write   Press BACK to review the regular verb endings.
```

Branches by the shifted NEXT and shifted BACK keys are controlled with the -next1- and -back1- commands. The tag is the unit to proceed to when SHIFT-NEXT or SHIFT-BACK is pressed. NEXT is the only function key which is always active. BACK, SHIFT-NEXT, and SHIFT-BACK are active only if there is a corresponding -back-, -next1-, or -back1- command in the unit. When function keys other than NEXT are active, it is a good practice to let the student know with a -write- statement what keys are available.

Units which the student reaches by pressing NEXT, BACK, SHIFT-NEXT or SHIFT-BACK are defined as main units. A main unit may contain several kinds of information: branching instructions, text and graphics, questions and commands to judge the student's response to the questions, instructions for the student about how to progress in the lesson. The TUTOR code to perform any one of these tasks can be written in a separate unit and that unit may be attached to the main unit with the -do- command. The tag of -do- is the name of the unit to be attached. When a unit is attached by -do-, its contents are treated as if they appeared in the main unit at the point where the -do- is located. After the commands in the attached unit are performed, the statements following the -do- are executed. There is no full screen erase when a unit is attached by -do-.

The code below is a portion of a lesson in which -do- is used in several main units to attach the message "Press BACK to review." In each unit in which the BACK key is active, unit "pback" is attached, displaying the text at location 3Ø18. If the author should decide to move that text to a different screen location, he would need to make only one change in unit "pback".

```
unit    first
next    second
        .
        .
        .
*
*
unit    second
next    third
back    first
do      pback
        .
        .
        .
*
*
unit    third
next    fourth
back    third
do      pback
        .
        .
        .
*
unit    pback
at      3Ø18
write   Press BACK to review.
*
*
```

Several advantages result from modularizing lessons by using -do-. The most obvious is the avoidance of repetition, saving the author's time and computer storage space. But more importantly, a modular design makes for easy readability of the program and facilitates correction of programming errors.

Below is a further illustration of the use of -do- in modular lesson design.

```
unit    part3
next    part4
back    part2
do      draw3
at      2816
write   Here is a square and a rectangle.
*
unit    part4
next    part5
back    part3
do      draw4
at      2312
write   Three Boxes!
*
*
*
unit    draw3
draw    1616;832;1648;2432;1616
draw    1216;1248;2Ø48;2Ø16;1216
*
unit    draw4
box     1912;1219
box     1925;1Ø32
box     1938;745
```

The branching structure of the lesson is readily apparent, so if an error in branching has been made, it is easy to find and correct. If there is an error in one of the drawings, it too can be quickly found in its own unit. Such readability is an important factor when the author returns to work on a lesson after several days away from it, or when two or more authors work together.

The factor that determines whether a unit is a main unit or an attached unit is how the student reached it. Units reached by pressing a function key are main units. Units reached by a -do- command are attached units. A student is always in a main unit; that main unit may attach other units which are treated by the computer as part of the current main unit.

More About Creating Displays

We have seen that the -at-, -write-, -draw-, and -box- commands are used to create displays. The tags of -draw- and -box- use screen addresses to determine the location of the drawings. The addresses may be in the line-character grid (coarse grid) or in the fine grid, which allows more precise specification of points on the screen.

The fine grid is a matrix of 512 dots by 512 dots. The lower left corner of the screen is location Ø,Ø. The upper left is Ø,511; lower right is 511,Ø; upper right is 511,511. When fine grid coordinates are used in the tags of commands, the x (horizontal) coordinate is given first, then the y (vertical) coordinate. The two are separated by a comma. For example, location 1ØØ,78 is 1ØØ dots from the left of the screen and 78 dots from the bottom. Figure 2.3 shows several fine grid screen locations.

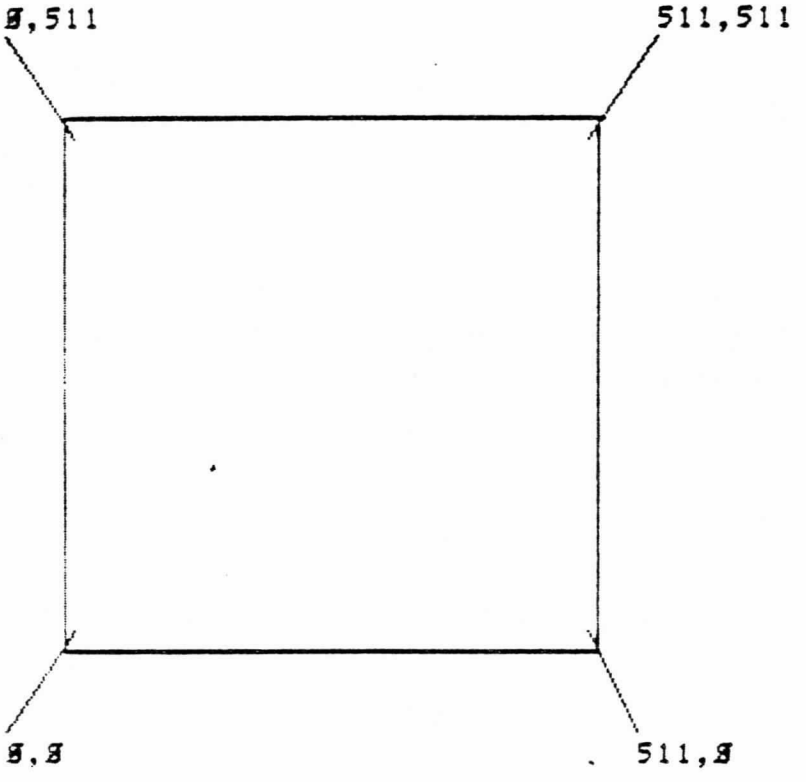


Figure 2.3 Fine Grid Screen Locations

The -draw- command can be used to create any figure composed of lines. The tag of the -draw- lists the points between which lines are to be drawn:

```
draw    1Ø1Ø;1Ø15;81Ø;1Ø1Ø
```

This statement draws a triangle with corners at locations 1Ø1Ø, 1Ø15, and 81Ø. The arguments in the tag must be separated by semi-colons.

Unconnected points within the figure are indicated by the argument "skip":

```
draw    1Ø1Ø;1Ø15;skip;1Ø22;1Ø28
```

A line will be drawn from location 1Ø1Ø to location 1Ø15, and another will be drawn from location 1Ø22 to location 1Ø28.

Both fine and coarse grid may be used in the tag of -draw-:

```
draw    32,12;4Ø,18
```

This command draws a line from the point 32 dots from the left and 12 dots from the bottom of the screen to the point 4Ø dots from the left and 18 dots from the bottom of the screen. Note that the two locations are separated by semi-colons while the x coordinate of each location is separated from the y coordinate by a comma.

Fine and coarse grid coordinates may be mixed in the tag:

```
draw     1617;2Ø2,377
```

The -box- command draws rectangles. The tag contains the screen locations of two opposite corners of the box, separated by a semi-colon. If the box is to be more than one line thick, a third argument stating the thickness can be added to the tag.

```
box      1418;1922
box      8Ø3;1115;3  $$ wall of box are 3 dots thick
box      264,333;1645
```

If the third argument is a positive number, the edge of the box is built up in an outward direction from the corners. If the thickness is a negative number the buildup is inward.

For drawing circles, the -circle- command is used. Its tag is the number of dots in the radius:

```
at      12Ø8
circle  12
```

This command draws a circle with a 12 dot radius whose center is at location 12Ø8. To draw an arc, use the 3-argument form of the -circle- command:

```
at      144,79
circle  97,42,138
```

This form of the -circle- command draws an arc. The second and third arguments specify the beginning and ending angles for the arc. They are given in degrees, but without a degree sign.

Three other commands, -pause-, -erase-, and -mode-, are also used in displays. The -pause- command may have several kinds of tags. The form

```
pause   keys=keyset
```

causes execution to stop until the student presses any key on the keyset. The form

```
pause   keys=next
```

halts execution until the NEXT key is pressed. More than one key can be listed in the tag:

```
pause   keys=next,back,back1
```

If keys other than those listed are pressed, they are ignored. Pressing NEXT causes execution of the unit to continue with the commands below the -pause-, but pressing any other listed key results in taking the branch specified for that command. For example:

```
unit    pr3
next    pr4
back    pr2
back1   printro
        .
        .
        .
pause   keys=next,back,back1
        .
        .
        .
```

If the student presses NEXT he continues in unit "pr3". If he presses BACK he is taken to unit "pr2" and if he presses SHIFT-BACK he is taken to unit "printro".

The -pause- command can be used effectively when a large amount of text is to be presented on the screen. One can display one paragraph, pause until the student signals that he is ready to go on, and then display a second paragraph. An example of this is shown in unit "hair".

```
unit    hair
at      5Ø9
write   Fashions in hairstyles seem to be a perennial
        topic of interest. From an article in the Sept.
        27, 1895 issue of the Daily Illini:
at      322Ø
write   Press NEXT to continue.
pause   keys=next
at      912
write   Perhaps the greatest peculiarity of the American
        college youth is his hair. The hair in itself is
        not so peculiar, although nearly every shade and
        color in the prism is represented, but the style
        in which it is worn. Its length varies from five
        to ten inches and is usually in a much tousled
        condition.
pause   keys=next
at      1712
write   The leading styles are the "duster" in which the
        hair is worn perpendicular to the tangent plane
        at the point of contact. Other styles are the
        "chrysanthemum," from which that popular flower
        derives its name, and the "mop" in which no two
        hairs are the same length.
```

The -pause- command is useful not only when large amounts of text are to be displayed; it can be used to emphasize several ideas by separating them, as in unit "tenses".

```
unit    tenses
at      202
write   The imperfect tense is used:
at      505
write   A:   To describe a habitual past action:
at      715
write   Quand j'étais jeune, je me levais à six heures.
        When I was young, I used to get up at 6 am.
pause   keys=keyset
at      1105
write   B:  To describe what was going on when another
            action took place:
at      1415
write   Il pleuvait quand j'ai quitté la maison.
        It was raining when I left the house.
pause   keys=keyset
at      1805
write   C:  To describe a situation which existed in
            the past:
at      2115
write   L'école n'était pas loin de ma maison.
        The school was not far from my house.
pause   keys=keyset
at      2505
write   D:  To describe how a person felt, looked, etc.
            in the past:
at      2817
write   J'espérais vous voir au bal samedi soir.
        I was hoping to see you at the dance Saturday
        evening.
```

The -erase- command erases the display. The commands

```
at      1010
erase   20
```

erase 20 character spaces beginning at location 1010. There is a two-argument tag for -erase- which erases a block of characters:

```
at      803
erase   12.4
```

This erases 12 characters starting at location 803 and continuing for 4 lines, erasing from 803 to 814, from 903 to 914, from 1003 to 1014, and from 1103 to 1114.

Another way to erase is to use the -mode- command. The -mode- command has three possible tags: write, erase, and rewrite. The terminal is normally in -mode write-. Executing a -mode erase- causes subsequent display commands to be erased rather than written; the dots of the screen which would be lighted for the display are instead turned off. When displays are put in attached units, the -mode- command can be used with -do- both to write and erase the display. Units "shapes" and "triangle" show such a use.

```
unit    shapes
box     1917;1141  $$ rectangle
do      triangle
at      2412
write   Press NEXT to make the triangle disappear.
pause   keys=keyset
mode    erase
do      triangle
at      2412
write   Press NEXT to make the triangle disappear.
**      erases the sentence
mode    write
at      2415
write   Now there's only a rectangle.
        .
        .
        .
unit    triangle
draw    1917;1129;1941
*
```

The first -do triangle- shows the display. After the -pause- unit "triangle" is done again, but this time it is erased. Note that you must reset the terminal mode to write with a -mode write- statement.

Displays executed in -mode erase- erase only those dots which form part of the display. In -mode rewrite- the entire screen area in question is erased before the new display is written.

Exercises

The first step in writing a lesson is to determine as completely as possible what the student will see on the screen and what actions will be taken by PLATO when the student presses keys on the keyboard. The next step is to detail the flow of control; i.e. the student's order of progression through the units upon pressing NEXT, BACK, or whatever branching keys you designate. A convenient way to do this is to sketch on paper the display for each unit, and to draw a flow chart showing the flow of control.

On paper, write a TUTOR program with three main units: title page, presentation 1, and presentation 2. (In the exercises for the next chapter, you will learn how actually to store and execute the program on the computer.) The title page should contain text and some simple graphics such as a box or underlining. The first presentation unit should present a drawing of any geometric figure (e.g. triangle, square, trapezoid) and a sentence or two about the drawing. The second presentation unit should display the same or a different drawing, then erase it, and show a sentence or two of text. The branching (flow of control) should be as shown in the flow chart in figure 2.4. Remember to tell the student what branching keys are available. Because your geometric figure will be used more than once, it should be in an attached unit. The actual text presented in your lesson need not be complex or extensive; concentrate on designing the structure of the program. Figure 2.5 shows one way to code part of this exercise. You may follow it exactly or use it as a general guide.

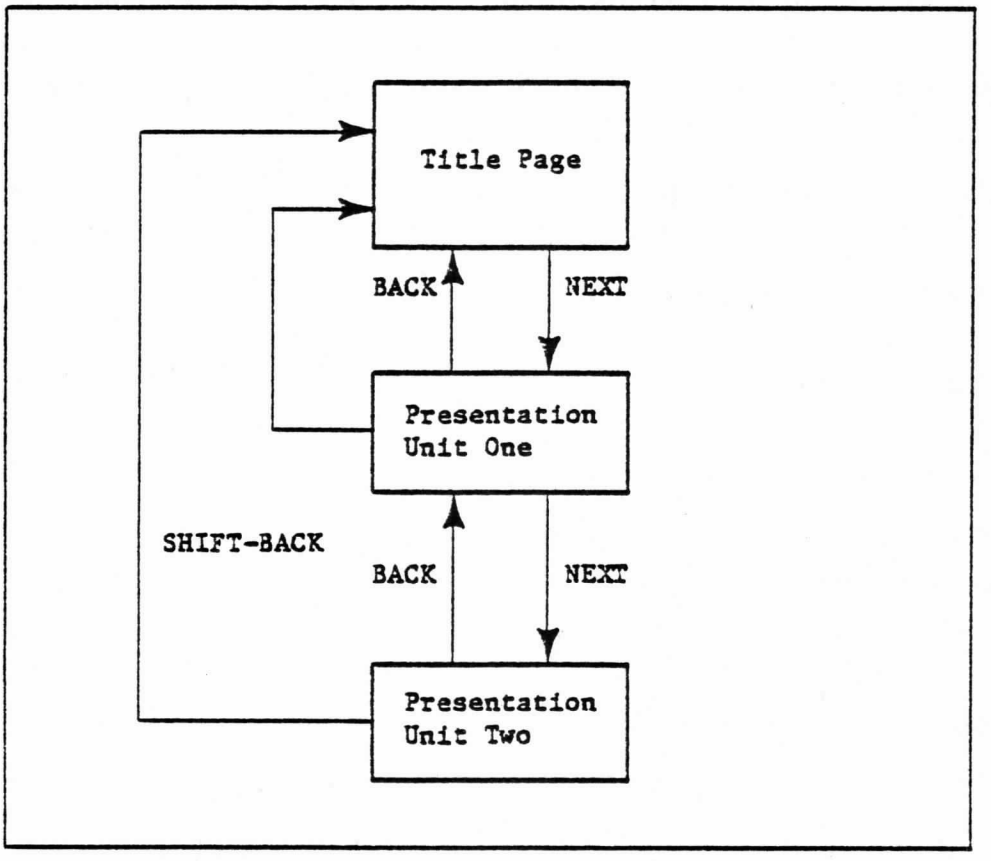


Figure 2.4 Flow Chart for Exercise 2

```
unit    title
next    rec1
at      1325
write   GEOMETRIC
         FIGURES
draw    1325;1334;skip;1433;1424
*
unit    rec1
next    rec2
back1   title
back    title
do      rec
at      2421
write   This is a rectangle.
*
        .
        .
        .
unit    rec
box     1819;1Ø43
        .
        .
        .
*
```

Figure 2.5 Sample code for Exercise 2

Chapter 3: Using the TUTOR Editor

At the Author Mode page, you may type the name of a lesson and then press DATA to run the lesson in student mode or NEXT to inspect or edit the TUTOR code of the lesson. Lessons are stored in files, which are areas of space on a magnetic disk. A file is divided into blocks. The first block contains identifying information about the file. The other blocks are where you type the TUTOR code of your lesson. Upon entering a newly created file, you are taken directly to the identifying information. If the file is not newly created, press DATA to see these information pages. Figures 3.1 through 3.5 show some representative DATA pages.

The author information page (figure 3.2) gives the name and address of the author and information about the lesson contents. The associated files page (figure 3.3) lists other files which are associated with your lesson. You should leave the -use- file and the processor lesson blank. The notes file is the file where comments made by users of your lesson are stored. This file is specified automatically when the file is created. It is the notes file used by all the lessons in your account. The codewords page (figure 3.4) allows you to limit who can edit and inspect the TUTOR code of the lesson. Every author has a slot in his records (in addition to the name, group, and password) into which he can type a security code. If the security code in this slot matches the change code of a lesson, the author may edit that lesson. If it matches the inspect code of a lesson, the author can inspect that lesson but cannot edit it. No one should edit a

lesson other than his own, even if its author has neglected to protect it with a change code. Codes 3 through 6 govern links between the lesson and other lessons--you may ignore them for the moment. The system personnel access flag permits PLATO systems programmers and consultants to inspect your file in case of malfunctioning of the system or when you request consulting help. You should leave this flag set to ALLOWED in order to receive prompt consulting help. The editing specifications page (figure 3.5) allows you to modify some of the normal conditions of the TUTOR editor. As a beginning author, you should leave these specifications as they are.

Pressing BACK one or two times while in the DATA section of the lesson takes you to the Block Display. This is the page you see upon entering a lesson every time except the first. The block named "*directory" contains the identifying information just discussed. To enter TUTOR code into the lesson, you must make a new block. You can make new blocks as needed, up to 7 blocks. Figure 3.6 shows the Block Display of a lesson. Within each block is TUTOR code. Figure 3.7 shows a block containing TUTOR code.

Never does an author write a complete lesson at once. He writes a small section, tests it by running it in student mode, corrects any errors, tests it again, and repeats that process until the section is perfect. Then he goes on to another section. To test TUTOR code that you have just written press SHIFT-STOP from either the Block Page or from within a block. This causes the lesson to be <u>condensed</u>, that is, prepared for use in student mode. Condensing is the process of translating a lesson from TUTOR into the machine language actually used by the computer to execute the lesson. Rarely does a lesson condense perfectly the first time; the author almost always has made typographical errors or errors in the syntax of the TUTOR commands. These must be corrected or the lesson will not execute properly. Figure 3.8 shows the display seen when a lesson has condense errors.

```
Lesson name ---- celia2                Disk pack -- florence
Starting date -- 85/12/74              Account ---- apso

Last edited ---- 85/11/78 13.44.32
          by ---- celia of pso
          at ---- 6-21

                Type the appropriate letter: >

                     a.  Author information

                     b.  Associated files

                     c.  Codewords

                     d.  Editing specifications

                        HELP available
```

Figure 3.1 Lesson DATA Page

```
Lesson name ---- celia2

Account -------- apso


Press the associated number to change an entry.

Author Information:
     1. Name ---------------  davis, celia
     2. Dept./Affiliation --  cerl
     3. Telephone number ---  3-6211

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

Figure 3.2 Author Information Page

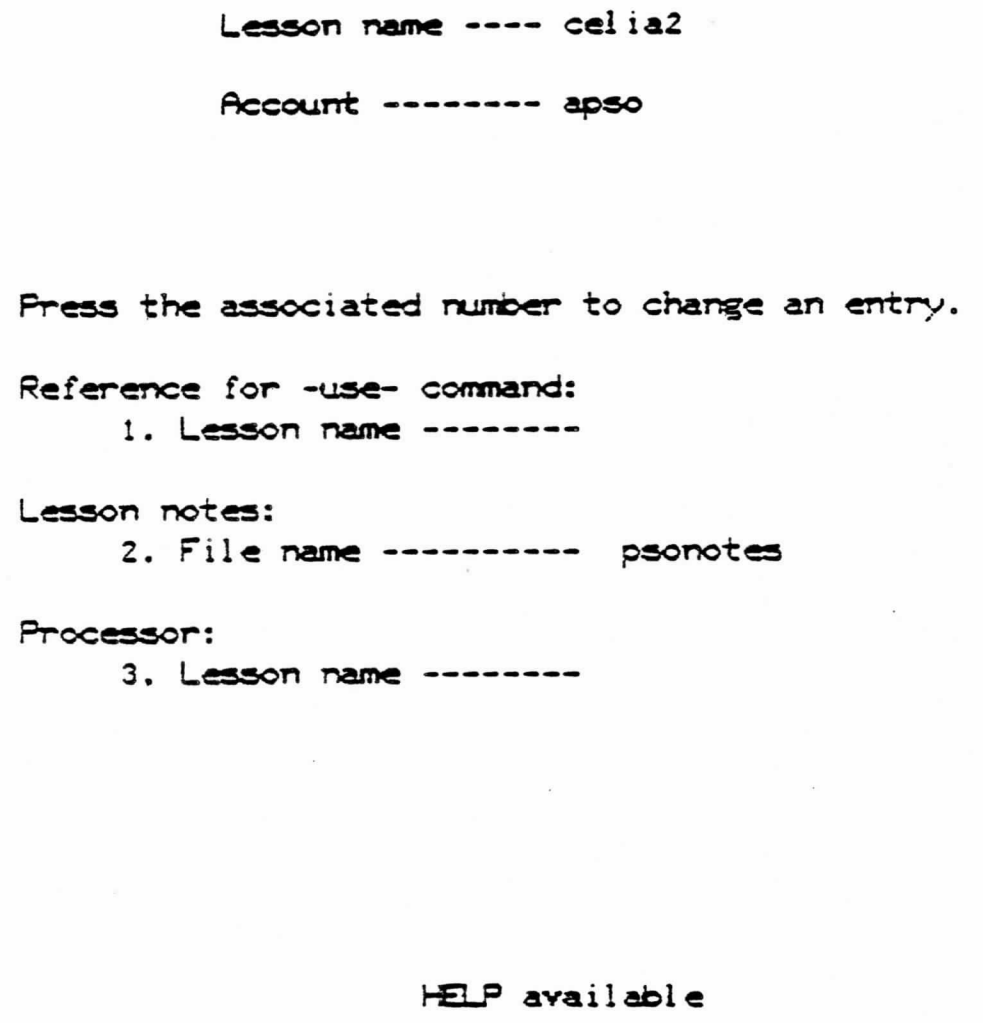


Figure 3.3 Associated Files Page

Lesson name ---- celia2

Account -------- apso

Press the associated number to change an entry.

SECURITY CODES:

1. To change lesson --- **********
2. To inspect lesson -- **********
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- file --- blank--OPEN TO ALL

Access to file by system personnel:

7. System Access ------ ALLOWED

Figure 3.4 Codewords Page

```
Lesson name ---- celia2

Account -------- apso


Press the associated letter to change an entry.

    Character set to be loaded:
        a. Lesson Name ----------
        b. Block Name -----------

    Microtable to be loaded:
        c. Lesson Name ----------
        d. Block Name -----------

    Display and typing control:
        e. Lines shown initially--   5
        f. When SPACE pressed ----  13
        g. Tab settings ----------  TUTOR

    Mod Word Options:
        h. Indicate changes ------  NO
        i. Save changes ----------  NO

    Type of print desired:
        j. Print option ----------  NORMAL

                    HELP available
```

Figure 3.5 Editing Specifications Page

LESSON--crd HELP available

PART 1 OF 2

BLOCK	NAME
a	*directory
b	def/title
c	begin
d	st1
e	wr1
f	wr4
g	wr7

Figure 3.6 Block Display

```
        BLOCK 1-e = wr1                    SPACE =  47
 1   unit     wr1
 2   do       helpers
 3   next     wr2
 4   *do       charles(296,336)
 5   do       charles(232,336)
 6   do       think
 7   at       2ØØ5
 8   write    Charles habite à Amiens.
 9   draw     2Ø13;2Ø19
1Ø   at       21Ø5
11   write    Charles
12   do       liner
13   write    à Amiens.
14   at       23Ø5
15   write    Il est très heureux
16   draw     23Ø8;2311
17   at       24Ø5
18   write    Il
19   do       liner
2Ø   write    très heureux
21   at       26Ø5
22   write    parce qu'il va rendre visite à son ami Hubert
23   draw     2617;2619
24   at       27Ø5
25   write    parce qu'il
26   do       liner
27   write    rendre visite à son ami Hubert
28   at       29Ø5
29   write    qui habite à Paris.
3Ø   draw     29Ø9;2915
31   at       3ØØ5
```

Figure 3.7 Line Display

```
Errors found when condensing lesson 'helping'
Errors: 1      Units: 2      Condensed Words: 75
```

1 undefined variable or bad define format test 1-e/2
calc name*2
Δ

More info on which error? >
SHIFT-NEXT to edit error

HELP is available
DATA to try lesson

Figure 3.8 Condense Errors Page

Exercises

1. Go into your lesson in the editor and fill in the identifying information. If your lesson is a new one, you will go directly to the proper page; if it is not new, press DATA to reach the page. To fill in an item, press the letter or number beside it, type the information and then press NEXT. Use the suggestions in this chapter about what to fill in.

2. Return to the Block Page of your lesson and press HELP. Figure 3.9 shows what this page looks like. Choose the section entitled "Introduction to Editing". After choosing this section you see an index similar to the one shown in figure 3.10. Start in section "a" and read all sections.

3. Enter some TUTOR code into the lesson (you may have to make a new block to do this). Use the units you wrote in the exercises for Chapter 2. Use the i directive to insert the TUTOR code. After entering the TUTOR code, find a line containing a typing error, and replace it using the r directive. Use the COPY or EDIT keys to correct the error. (If you made no typing errors, choose a correct line to replace for practice.) Return to the block page, then re-enter the block. Pactice using the space bar to display more lines. Use the f, b, u, and U directives until you are familiar with their operation. Find a line whih should be deleted, bring it to the top with the f directive, and delete it. Then insert a new line to take its place. (Again, if you have no line with mistakes, choose a good line and practice deletion using it.) Learning to use the TUTOR editor can be very frustrating, so if you have trouble, ask your group director for help or call a consultant on TERM-consult. Figure 3.11 lists some frequently used editing directives.

4. Condense the lesson. If there are any condense errors, return to the lesson and correct them, then condense again. When all errors are corrected, go through the lesson in student mode and verify that it works as you expect. If it doesn't, note carefully what is wrong. Return to editing and see if you can determine the cause of the errors and correct them. Then test the lesson again. Repeat the process until it works perfectly. If you need help correcting either condense errors or mistakes in the running of the lesson, call a consultant on TERM-consult.

Editing HELP

1. Introduction to Editing
   (What you need to know to get started)

2. Block Directory Options
   (Options available on the page listing
   the names of the blocks)

3. Line Display Options
   (Options available within the blocks)

4. Mod Word Options

Figure 3.9 Editor HELP Page

Introduction to Editing

Press the letter of your choice

a. Blocks and Parts

b. The Line Display

c. Format of Editing Directives

d. Moving Around in the Line Display

e. Inserting, Replacing & Deleting Lines

f. Moving Lines from One Location to Another

g. Creating Displays Automatically

SHIFT-BACK for Main Help Index

Figure 3.10 More Editor HELP

To enter a block:
press the letter of the block
To create a block:
press the shifted letter of the block after which you want to create a new one. To create block b, press capital A.
To change the name of a block:
go into the block and press LAB.

Frequently used editing directives:

f	move forward
b	move backward
u	move to next unit
U	move to previous unit
i	insert lines
r	replace lines
d	delete lines
s	save lines
is	insert the saved lines
id	insert a display
sd	show the display, then change to id
space	show more lines
out	leave block, ignoring changes

Figure 3.11 Some Editing Directives

Chapter 4: Judging Student Input

Computer-based education is most effective when there is a high degree of interaction between the student and the lesson material. PLATO lessons can be highly interactive because the TUTOR language allows for sophisticated interpretation of student input and for branching based on student input. The -arrow- command allows student input. It does two things: plots an arrowhead at the location given in its tag, and signals the computer that a response is expected. Execution of the program then stops until the student presses NEXT. When NEXT is pressed, judging of the student's response begins. The -answer- command is one command used to judge the correctness of the student's response. Its tag is the correct answer. The unit below shows a simple question and answer.

```
unit    states
next    hawaii
at      1218
write   Which state was the forty-ninth to be admitted to the Union?
arrow   142Ø
answer  Alaska
write   Mt. McKinley is there.
```

When this unit is executed, the question is displayed on the twelfth line, 18 spaces from the left. The -arrow- command does two things: it plots an arrowhead at the location given in the tag (14 lines down, 2Ø spaces over) and it signals the computer that a response is expected. When the student

types his response, it will appear on the panel next to the arrowhead. When the computer encounters the -arrow- command, execution of the unit stops until the student enters his response and presses NEXT. After the student presses NEXT, his response is compared to the tag of the -answer- command. If the student has entered "Alaska", the -write- statement below the -answer- command is executed, that is, the phrase "Mt. McKinley is there" is displayed, and the word "ok" is displayed beside the student's response. If the student's response does not match the tag "Alaska", the word "no" is displayed beside his response. He must erase his incorrect response and enter another one. He cannot proceed to unit "hawaii" until he has satisfactorily completed the question.

When a student answers incorrectly, it is desirable to display a message that will guide him toward the right response. This can be done with the -no- and -write- commands.

```
unit    states
next    hawaii
at      1218
write   Which state was the forty-ninth to be admitted to the Union?
arrow   1420
answer  Alaska
write   Mt. McKinley is there.
no
write   It was known as Seward's Folly.
```

With these additions to unit "states", the student will receive more help than simply the "no" judgment beside his incorrect answer. For any answer other than "Alaska", the sentence "It was known as Seward's Folly" will be displayed. When the student erases his incorrect answer, the tag of the -write- command is also erased. If the student answers correctly, the -write- below the -no- is not displayed.

Commands which depend for their execution on matching the preceding -answer- or -no- command are called answer-contingent or no-contingent commands. They are executed only when the -answer- or -no- with which they are associated is matched. Answer-contingent and no-contingent -write- statements need not be preceded by an -at- command; they are automatically placed three lines below the arrow. If a different location is desired, an -at- statement can be inserted above the -write- statement.

Unit "states" accepts as correct only the response "Alaska". To allow for responses like "The state of Alaska" or "Alaska was the 49th state

admitted", some modifications to the -answer- tag can be made. Synonyms, any one of which is correct, can be specified in an -answer- tag by enclosing them in parentheses; and "ignorable" words, i.e. words whose presence or absence is immaterial, may be enclosed in < > brackets.

```
unit    states
next    hawaii
at      1218
write   Which state was the forty-ninth to be admitted to the Union?
arrow   142Ø
answer  <it,is,was,it's,the,to,be,admitted,state,forty-ninth,49th> Alaska
write    Mt. McKinley is there.
no
write   It was known as Seward's Folly.
```

With these additions to the -answer- tag, the student has more leeway in his answer. Any word which appears within the angle brackets is ignored if it occurs in the student's answer. Responses such as "It's Alaska." or "Alaska was the forty-ninth state admitted" are all judged "ok" and cause the answer-contingent -write- statement to be done.

If capitalization is unimportant in the student's response, it may be so indicated by use of the -specs bumpshift- statement. The -specs- command sets certain specifications for interpreting the response. The tag "bumpshift" means that all shifted keys (capital letters are shifted keys) will be treated as if they were not shifted, making "A" and "a", for instance, equivalent. When -specs bumpshift- is used, no shifted keys may appear in the tag of the -answer-, even though they can be used by the student.

```
unit    states
next    hawaii
at      1218
write   Which state was the forty-ninth to be admitted to the Union?
arrow   142Ø
specs   bumpshift
answer  <it,is,was,it7s,the,to,be,admitted,state,forty-ninth,49th> alaska
write   Mt. McKinley is there.
no
write   It was known as Seward's Folly.
```

In this example, the word "Alaska" in the -answer- tag was changed to "alaska". Note that "it's" in the list of ignorable words must be changed to "it7s" because the apostrophe is the shifted "7" key.

Spaces and punctuation marks in the student's response are all treated as word separators. Punctuation cannot appear in the tag of the -answer- command, but the student may use any punctuation and may use two or more spaces between words. TUTOR has commands to handle cases in which exact punctuation by the student is required.

Incorrect responses which are reasonably close to the tag of the -answer- are automatically marked for the student. Misspelled words are underlined ($\underline{\text{xxxxx}}$). Words out of order are marked with an arrow (⇆). Words in the response which do not occur at all in the -answer- tag are underscored with x ($\underline{\text{xxxxx}}$). Missing words are indicated by the symbol Δ .

The -wrong- command is used for matching specific incorrect responses. A wrong-contingent -write- statement can give the student a comment appropriate to his particular incorrect answer.

```
unit     states
next     hawaii
at       1218
write    Which state was the forty-ninth to be admitted to the Union?
arrow    142Ø
specs    bumpshift
answer   <it,is,was,it7s,the,to,be,admitted,state,forty-ninth,49th> alaska
write     Mt. McKinley is there.
wrong     <it,is,was,it7s,the,to,be,admitted,state,forty-ninth,49th> hawaii
write     Hawaii was the fiftieth state to be admitted.
no
write    It was known as Seward's Folly.
```

Like -answer-, -wrong- statements may specify synonyms and ignorable words. If the student enters "Hawaii" or any acceptable match to the tag of the -wrong- command, the sentence "Hawaii was the fiftieth state admitted." is displayed; and the "no" judgment appears on the panel next to his response. All -answer- and -wrong- commands must come before the -no- command because the -no- command causes any response not previously matched to be judged "no" and the no-contingent -write- statement to be done. The -no- command thus provides a way to display a comment when the student's answer matches none of the expected correct or incorrect responses.

If groups of words in a student's answer are to be treated as a phrase, this is done by separating the words by asterisks in the tag:

```
answer  He (looked*up*to,revered) <the> (teacher,philosopher,Aristotle)
```

The phrase "looked up to" is synonomous with the word "revered". If the student makes errors in the phrase, they are marked by a line of asterisks (******) below the phrase.

An -arrow- may have more than one -answer- or -wrong- command associated with it. Sometimes it is possible to allow for different responses by the use of synonyms. When this is not possible or when different comments should be displayed for different responses, two or more -answer- or -wrong- statements should be used.

```
                .
                .
                .
at              805
write           Which Hawthorne novel that we have studied explores the theme
                of conflict between social tradition and self-expression?
arrow           1307
answer          <The,the> Marble Faun
write           The Scarlet Letter is another example.
answer          <The,the> Scarlet Letter
write           The Marble Faun is another example.
```

Because a different comment is to be written for each right answer, two -answer- commands are used.

Student branching may be tied to -answer- commands. In an index unit, for example, one might present several topics and allow the student to choose which he wishes to study. Such branching can be done with the -jump- command. The -jump- command causes an immediate branch to the unit named in its tag. Any commands placed below the -jump- are not done; the student proceeds immediately to the unit named in the tag of the -jump-. The unit below illustrates an index using answer-contingent -jump- commands.

```
unit            index
at              1014
write           Choose a topic:
at              1322
write           a.  Gerunds
                b.  Nouns in Apposition
                c.  Relative Clauses
arrow           1030
answer          a
jump            gerunds
answer          b
jump            appos
answer          c
jump            rclause
no
write           Choose a, b, or c.
```

If the student answers "a" he is sent immediately to unit "gerunds". If he answers "b" he is branched to unit "appos", and to unit "rclause" if he answers "c". Any response other than "a", "b", or "c" is judged "no" and the comment "Please choose a, b, or c" is displayed. A unit reached by a -jump- command, like those reached by pressing a function key, is a main unit.

TUTOR commands are divided into two categories, judging commands and regular commands. Commands which are used when evaluating a student response are called judging commands; all others are regular commands. Regular commands include -unit-, -next- and the other key branching commands, -at-, -write-, and -arrow-. Judging commands include -specs-, -answer-, -wrong-, and -no-. Correspondingly, the PLATO computer has two states, regular state and judge state. When the computer is in regular state, only regular commands are executed; when in judging state, only judging commands are executed.

PLATO begins a unit in regular state. All regular commands are done until an -arrow- command is encountered. The location of the -arrow- is noted, and any regular commands below the -arrow- and above the first judging command are done. When a judging command is encountered, PLATO stops and waits for the student to enter a response. When the student presses NEXT, judging state begins. The student's response is compared to the tag of the first judging command. If they match, judging state ends, and any regular commands after the matched judging command and before the next judging command are done. An "ok" or "no" is displayed on the panel, and the student continues in the unit if the judgment was "ok" or returns to the arrow if the judgment was "no". If the student's response does not match the tag of the first judging command, judging state continues. Any regular commands below an unmatched judging command are skipped. If, by the end of the unit, no judging commands have been matched, regular state is returned to and a "no" judgment is made. The student goes back to the -arrow- and enters another response. Judging state begins again. This process continues until an "ok" judgment has been received.

The -endarrow- command delimits the portion of a unit relevant to the response judging for the preceding -arrow-. It is useful when some regular commands are to be executed no matter which -answer- command was matched or when the unit contains more than one -arrow-.

```
          .
          .
          .
at        805
write     Which Hawthorne novel that we have studied explores the theme
          of conflict between social tradition and self-expression?
arrow     1307
answer    <The,the> Marble Faun
write     The Scarlet Letter is another example.
answer    <The,the> Scarlet Letter
write     The Marble Faun is another example.
no
write     Press BACK for a list of Hawthorne's novels.
endarrow
at        2010
write     This theme is central to ...
          .
          .
```

The -endarrow- marks the end of the judging commands for the preceding arrow. The regular commands below the -endarrow- are not done until the -arrow- is satisfied with an "ok" judgment. After the judgment the last -write- statement is displayed, regardless of which -answer- command was matched. The -endarrow- is a regular command. Figure 4.1 illustrates in a flow diagram the execution of TUTOR judging.

If the student's response is to be completely numerical, the judging commands -ansv- and -wrongv- can be used. The tag is a number or an expression.

```
unit      distance
at
write     What is the distance, in miles, between
          New York and Los Angeles?
arrow
ansv      3000
write     Too far to walk!
wrongv    1500
write     That's about half the distance.
no
write     It's roughly 3000.
```

If you want to allow some leeway with -ansv- and -wrongv-, you may include a second argument in the tag. This argument is a tolerance. It may be a percent or an expression. The statement

```
ansv      3000,10%
```

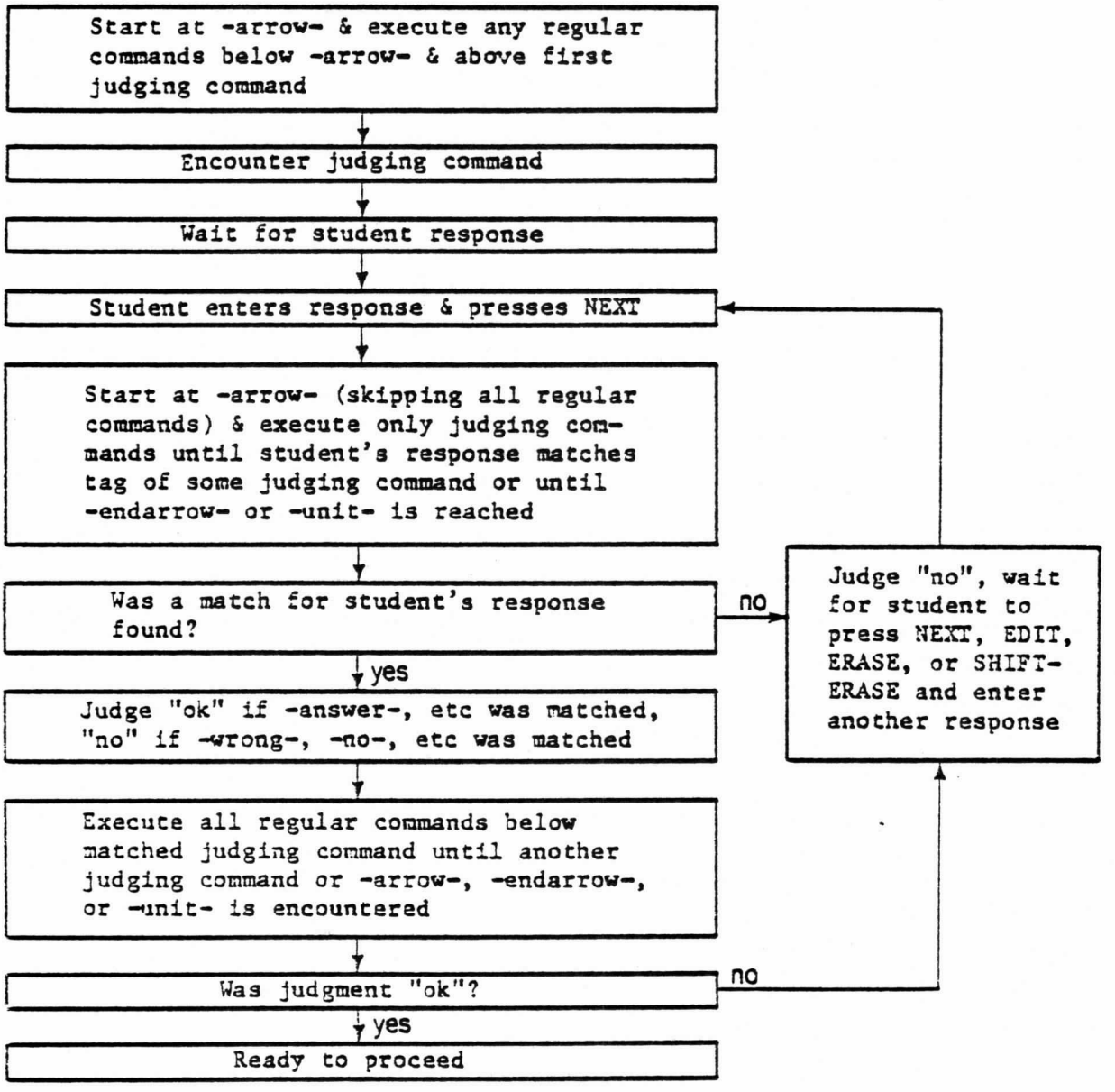


Figure 4.1 Operation of a TUTOR Unit

will be matched by any student response between 2700 and 3300. The statement

```
wrongv   1500,100
```

will be matched by any number from 1400 to 1600.

When a student enters a numerical response, PLATO evaluates the response and then compares the value of the expression to the value of the -ansv- or -wrongv- tag, so they are matched by an response that is mathematically equivalent to the tag. Student responses of "1500x2", "2800+200", "3x10^3" will all match the tag of the -ansv- above.

Exercises

Add to your existing program a new section of two question units. This section should have its own title page. In addition, add an index unit at the beginning of the lesson from which the student can choose either the geometry section or the new section. For subject matter we will use information about TUTOR. After you complete this exercise, your lesson will have the flow of control shown in figure 4.2.

The index unit should present two choices, geometry or TUTOR. Have the student press a letter to indicate his choice. Use the -jump- command to jump to the appropriate unit. The code on page 53 can be adapted for your index unit.

The title unit for the question section should simply state the topic. Use any graphics that you like. NEXT from this unit should go to question unit one, and BACK to the index unit.

The first question unit should ask the student to name the command used for attaching units to main units. In designing the tag of your -answer- command, allow enough leeway so that responses like "do" and "the do command" are judged ok. If the student doesn't get the right answer, display a hint (use the -no- command followed by a -write- for this). NEXT from this unit should go to the second question unit, and BACK to the title unit for the questions section.

The second question unit will ask the student to define the term "main unit". Write your -answer- tag to allow a range of responses such as "any unit reached by pressing a function key" or "the definition of main unit is a unit reached by jump or a function key" and so forth. Use angle brackets to indicate ignorable words. Provide a comment for wrong responses. NEXT from this unit should return to the index, BACK to the previous question.

Modify the last unit of the geometry section so that NEXT goes to the index.

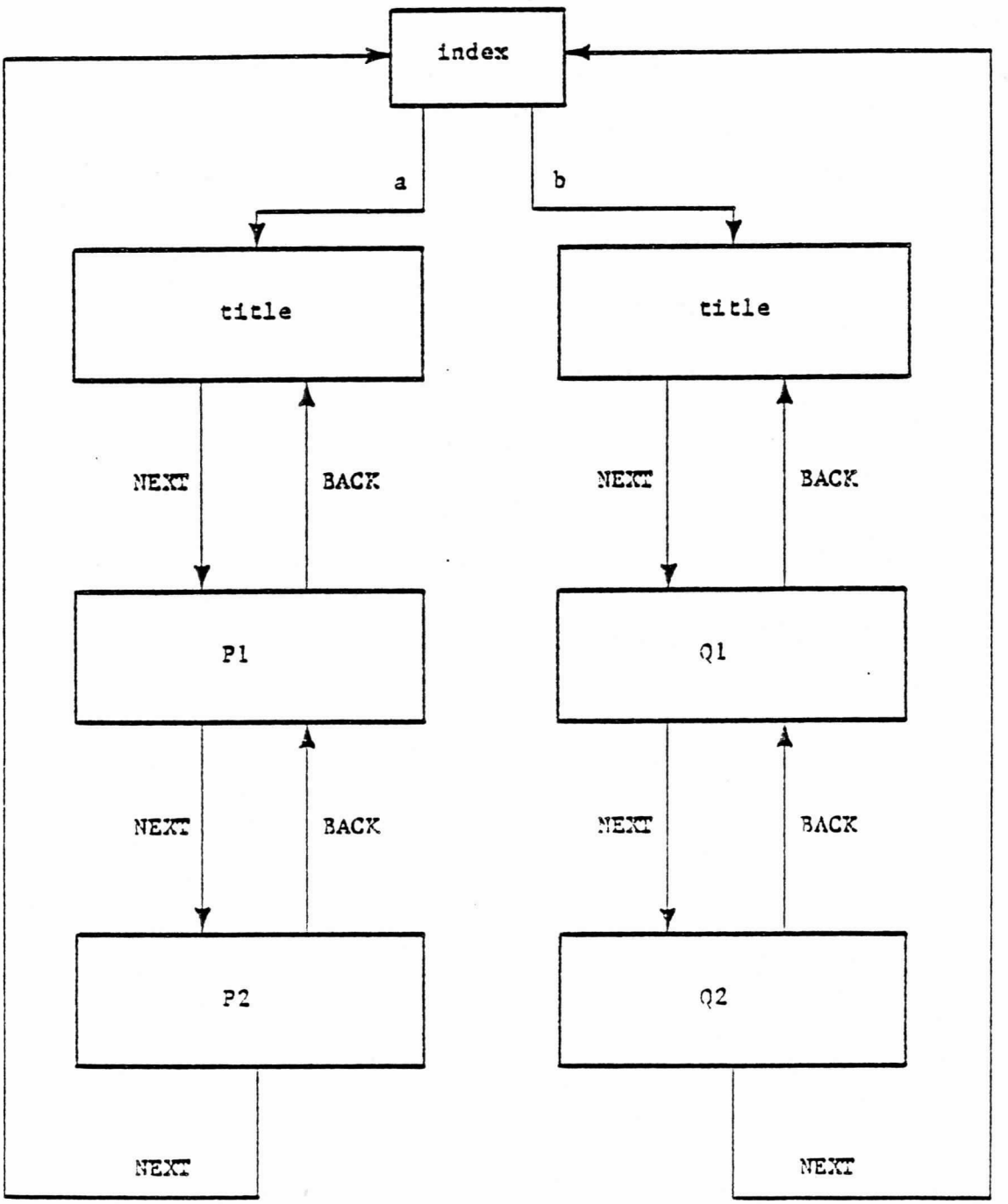


Figure 4.2 Flow Chart of Exercise

Chapter 5: Variables

Within the space of a very few chapters we have seen the first steps toward utilizing a vast computer system to provide educational materials quite unlike any heretofore possible. With only a few commands we have a means of creating high quality displays. We have seen how to pass some control of the learning environment to the student by means of student-initiated branching. We have begun to form a basis for individualized instruction through the immediate feedback available when judging student responses. But much more is possible when we explictly avail ourselves of one of the most obvious powers of the computer: the ability to compute.

PLATO is certainly capable of carrying out vast and complex computations; yet for a great many teaching strategies the only computations necessary are simple counting and occasional arithmetic. For example, after a student's second attempt at answering a question you may wish to supply a hint; after a third failed attempt you may want to branch the student to remedial material. Or instead of specifying all screen locations for displays at the time of writing a lesson, you might wish to calculate the locations for certain displays at the time the student is running the lesson. lesson.

For whatever computation is to be performed, the results must be stored somewhere so they will be available for later use. To this end, PLATO provides a means of storing results generated as a user executes any lesson. PLATO devotes a separate section of memory to each user to keep track of

that user's name and group, user type, the name of the lesson currently being run, the main unit being executed, and a variety of other data. Within that section of memory are 15Ø "memory slots" which can be used by the lesson being run to store numerical values. These spaces are called "student variables", a term which requires some explanation.

The designation "student" has two applications, one referring to a type of user and another referring to a mode of interaction with a given lesson. Our user type might be "author" but when we run, or execute, a lesson, we are said to be running it in student mode; the memory slots alloted to us can be used by the lesson to store values. Thus all types of PLATO users, author and instructors as well as students, have 15Ø student variables.

In the PLATO context, a variable is nothing more than a memory location, a portion of computer hardware that can store a number. A PLATO variable may be likened to the display unit of a hand calculator. The display unit can contain a number, positive or negative, up to a specific size. If the hand calculator has a memory, with a keypress one can store the displayed number in an internal memory location, a piece of calculator hardware. On PLATO, when appropriate commands are executed, numbers can be stored internally in pieces of computer hardware. PLATO authors commonly refer to these electronic devices as memory locations, "words" of memory, or simply variables.

The concept of the student variable is an important component of the great potential PLATO has for providing individualized instruction. A large number of students may all be running a single lesson, but the lesson can behave in as many different ways as there are students. The lesson orchestrates the storing and use of the contents of each student's student variables in a general way. The exact contents of student variables belonging to any one student will determine how we choose to use that individual student's response to the lesson at any given point in the lesson's execution. The structure of TUTOR allows us to write a lesson as though it were in communication with only a single set of student variables belonging to some single generalized student, without having to worry about the fact that many students might be running the lesson at once.

In writing lessons we can directly refer to a specific student variable by its "address". For example, we might choose to store a quiz score in

student variable 2Ø. Later in the lesson we can have a command which instructs PLATO to display the contents of student variable 2Ø on the screen. There are no rules or restrictions concerning which of the 15Ø variables we may use. It is not necessary to start with student variable 1 or to use them sequentially. There is, however, one step required in addition to designating the address of the student variable: we must indicate what notation system to use. PLATO, like most other computers, has two different methods of storing numbers in memory locations. One form, or format, is used explicitly to store integers, the numbers Ø, 1, 2, 3, ... The other format is a variation of scientific notation which permits the storing of numbers with fractional parts such as 14.5 or Ø.ØØ125 or extremely large or small numbers such as $3.Øx1Ø^{1Ø}$ or $6.62x1Ø^{-27}$. To specify integer format, add the prefix "n" to the student variable address. Thus a reference to variable n75 specifies at once the student variable number 75 and integer formatting. The other format is indicated by using the prefix "v". A reference to variable v75 refers to the same student variable, number 75, but specifies this alternate form of storage.

The prefixes n and v are not abbreviations for any words and are best considered as arbitrarily chosen letters. N variables are also called "integer variables". There is no corresponding term for v variables, but numbers stored in v variables are sometimes termed "floating point numbers".

Once we are accustomed to thinking in terms of n variables and v variables, it is easy to fall into the error of thinking that there exist two distinct sets of variables: this is not so. There are only 15Ø student variables. The arrangement of digits in any variable may be interpreted two ways, either as an integer or as a floating point number. The n or v designation specifies which of the interpretations is to be used.

Assigning Values to the Student Variables

One of the most powerful commands in the TUTOR language is the -calc- command. It is used to store a number in a variable:

```
calc    n1← 32
```

The statement is read "n1 is assigned 32." The integer 32 is stored in

student variable number 1 in integer, or "n", format. The symbol "← " is called the assignment arrow. It designates the operation of taking the numerical value of the right hand side of the symbol and storing that value in the variable on the left hand side in the indicated format. The previous contents of the variable specified on the left side of the assignment arrow are lost. In effect, the previous contents are erased and a new value is written into the memory location.

Any number, variable, or expression may be used on the right hand side of the assignment arrow:

```
calc    n2Ø← n1ØØ
```

Here the numerical value of the right hand side of the "← " is the contents of student variable 1ØØ interpreted as an integer. That value is stored in variable 2Ø, also as an integer. The contents of n1ØØ are not changed. The process of copying a value out of a variable does not disturb the original contents of that variable.

As might be suspected, the -calc- command allows us to perform calculations.

```
calc    v37← v1Ø1+v1Ø2
```

The contents of student variables 1Ø1 and 1Ø2, interpreted as floating point numbers, are added together and the sum is stored in v37. The original contents of v1Ø1 and v1Ø2 are undisturbed. The original content of v37 is overwritten with the new value, the sum.

```
calc    n5Ø← n5Ø+1
```

In this example, n5Ø is being used as a counter. The value in n5Ø just before the execution of the -calc- is incremented by one as the command is executed. The result is stored in the same student variable, n5Ø.

Besides addition, the operations of subtraction, multiplication, division, and exponentiation (as in $1Ø^3$) may be performed. To indicate multiplication one may use the symbol x (not the letter x). The key used to

type the multiply sign is located near the lower left of the keyboard. Alternatively, the -calc- command will also interpret the symbol * as denoting multiplication; this is an old convention still used by some other computer systems that do not have keysets containing a multiply symbol. For division, the symbols ÷ and / may both be used. To type a superscript to indicate exponentiation, press the SUPER key before typing the digit of the exponent. If the exponent has more than one digit you may either press SUPER before each digit, or you may press SHIFT-SUPER, type all the digits, and then press SHIFT-SUB to return to the normal line. The old convention of typing two asterisks to indicate exponentiation may also be used, as in 3**2, equivalent to 3^2.

Expressions are evaluated following order of operations rules. For an expression without parentheses, any exponentiation is done first, then all multiplcation, then division, and addition and subtraction last. For addition and subtraction, neither has precedence over the other. These operations are done in left-to-right fashion yielding the same result that would be obtained if one decided arbitrarily to give one operation precedence over the other. Let us examine an example:

```
calc    n12 ← 5 + 3 x n1Ø
```

If n1Ø contains the value 2, the expression to the right of the assignment arrow will be evaluated as 11. Notice that this is quite different from evaluating the expression straightforwardly from left to right. If the latter were done, one would start with 5, add 3 to it to get 8, and then multiply by 2, giving 16. If parentheses are present, the expression within the parentheses is completely evaluated following the same operation rules, before continuing.

```
calc    n12← (5+3)xn1Ø
```

Here the expression (5+3) is evaluated first; the resulting equivalent expression, 8 x n1Ø, is evaluated next. Sets of extra parentheses can be incorporated in any TUTOR expression to ensure that what is written is evaluated the way the author intended. Figure 5.1 summarizes the order of operations in TUTOR.

Order of Operations		Example Expression	Value of Expression
First	exponentiation	4^2	16
↓	multiplication	16x2	32
↓	division	1ØØ 25	2Ø
Last	addition, subraction (from left to right)	14-7+2	9
		12+8-3	17

Figure 5.1 Order of Operations in TUTOR

According to strict mathematics, multiplication and division are of equal precedence and should be done in left to right fashion. Some computer languages follow this rule. According to standard American printing conventions, expressions written on a single line are expected to be interpreted as if multiplication had precedence over division. TUTOR follows this latter convention. Thus, the TUTOR statement

```
calc    v2Ø ← 3 / 4 x v3Ø
```

is evaluated as 3 divided by the product of 4 and the contents of v3Ø, and not 3/4 times the value of v3Ø.

Should we have several calculations in a row to perform, we may avail ourselves of a convenience built into TUTOR. After the first -calc- statement, we need not repeat the command for any immediately succeeding -calc- statement.

```
calc    n2← n2+1
        n3← n3+1   $$ a new calculation without "calc" typed
        n4← n4+1   $$ another new calculation
at      512
write   Finished incrementing counters.
calc    n47 ← 1ØØ + 5 x n4   $$ "calc" must be typed here
```

The arithmetic expression on the righthand side of an assignment arrow may contain both n variables and v variables. The fact that numbers might have been stored in different formats presents no problem for PLATO because for calculation purposes retrieved n variable values are converted to v

variable format before use in the calculation. If we try to store the results of our calculation in an n variable, that is we designate integer format for storing, the value of the expression will be rounded to the nearest integer:

```
calc    n2Ø ← 49
        v3Ø ← 1ØØ
        n4Ø ← 1.ØØ + n2Ø / v3Ø $$ expression = 1.49; n4Ø contains 1
        n41 ← 1.Ø1 + n2Ø / v3Ø $$ expression = 1.5Ø; n41 contains 2
```

Naming Student Variables

When a lesson takes on any complexity, refering to the student variables by their "primitive" names, n1, n37, v12,... can become a source of confusion and mistakes. The values kept in student variables represent certain quantities having meaning to the lesson author, so it is desirable to give them names suggestive of their function. The -define- command associates a name with each variable used.

```
define  score=n1Ø,wrongs=n11,total=n2Ø,percent=v3Ø,
        quiz1=n4Ø,quiz2=n41
        .
        .
        .
calc    score← total-wrongs
        .
        .
        .
calc    percent← (1ØØ.Øxscore)/total
        .
        .
        .
```

You may name the variables in any way that is sensible to you. Defined names must begin with a letter and must not be more than 7 characters long. The name may contain digits in addition to letters. A single -define- command can be used to define a long list of variables, extending to many lines if necessary. Notice that for continuation lines of a -define- statement the command portion of the line is left blank and the continuation appears in the tag position. Note also that several definitions may occur on a single line provided that they are separated by commas. The symbol = should be interpreted as "is equivalent to". The defined name may be used

only after the point in the lesson where the -define- appears. In the above example, when PLATO reaches the line

```
calc    score← total-wrongs
```

it refers to the previously occuring -define- to determine which student variables are intended by the names "score", "total", and "wrongs". The calculation performed is the same as if one had written

```
calc    n1Ø← n2Ø-n11
```

Clearly, by carefully selecting names for the student variables used, an author makes his lesson much easier to read and removes for himself a source of mistakes in writing the lesson.

A highly recommended practice is to place the -define- command at the very beginning of a lesson. As you continue developing a lesson and find that you need to use more variables, go back to the beginning and add the new variable definitions to the original list. As a general rule of good programming, all definitions should be kept together under a single -define- command at the beginning of the lesson.

Displaying N Variables

Because the value in a student variable can change during the execution of a lesson, a dynamic means of showing the current value is needed. The -show- and -showt- commands serve this function:

```
define  score=n1Ø,...
        .
        .
        .
at      1Ø1Ø
show    score
```

These statements disply the contents of the variable "score" in integer format starting at location 1Ø1Ø.

Quite frequently, it is desirable to display the value of a variable in the middle of some text. This could be done with two -write- statements and an intervening -show- or -showt-, but a more convenient way is to "embed" the -show- or -showt- within the tag of the -write- statement:

```
at      27Ø5
write   You have answered ⟨show,corrects⟩ items
        correctly on the first attempt.
```

If "corrects" contained the value 12, this TUTOR statement would appear on the screen as "You have answered 12 items correctly on the first attempt." The embedded -show- can be abbreviated: ⟨s,corrects⟩. Embedding -show- commands within -write- commands has two advantages: the lesson code resembles more closely the actual display seen by the student, and the statement is treated as one statement for erasing purposes. Recall that when a student presses NEXT or ERASE after an incorrect answer, the tag of the last (and only the last) -write- statement is erased. A -write- with an embedded -show- is completely erased in such a case.

A glance at the keyset shows that the embed symbols are not part of the standard character set. These are two of a set of special symbols. To gain access to them, press the shifted □ key and then the key corresponding to the special character. The □ key is located to the right of the HELP key. SHIFT-□ is called the ACCESS key. To type the embed symbols press ACCESS, then Ø for ⟨, and ACCESS, then 1 for ⟩.

For displaying integers within a line of text the -show- command accomplishes the task perfectly. The leftmost digit of the integer appears at the explicity or implicitly specified screen location so that it is not necessary to account for the possiblility that the integer might be only one digit long in some cases, two digits long in others, and so forth. There are other applications, however, in which the position of the rightmost digit, rather than the leftmost, needs to be specified. The creation of a table of numbers is such a case. For a pleasing display one might want all integers in a column to have their last digits aligned. An alternative to the -show- command exists for this purpose. The -showt- command, or "tabulated show", positions the integers at the righthand side of a field of character positions. If the length of the character field is not explicitly

stated, PLATO assumes a length of 8 characters. If desired, an alternate field length can be specified in a second argument of the tag.

```
define  count=n37,item=n38,...
        .
        .
        .
at      510
showt   count  $$ rightmost digit at position 517
        .
        .
        .
at      614
showt   item,4  $$ rightmost digit at position 617
```

For the first -showt-, the first character position is at screen location 510, the second at 511, and so on with last, the eighth, position at 517. Should the number of digits in the value to be displayed be less than the number of character positions in the field, PLATO displays spaces before the number. If the number is too large to fit in the number of spaces you have provided, the field will be filled with asterisks to indicate this.

The -showt- command also has an embedded form which can be used to display labeled table entries.

```
define  pop1=n20,pop2=n21,pop3=n22,...
        .
        .
        .
at      410
write   Current Population Distribution for Model City
at      618
write   Sector 1 ‹showt,pop1,8›

        Sector 2 ‹showt,pop2,8›

        Sector 3 ‹showt,pop3,8›
        .
        .
        .
```

The rightmost digits of the three population figures will be aligned. Like the -show- command, -showt- has an abbreviated form for use in embedded form: ‹t,pop1,8› is interpreted to mean ‹showt,pop1,8›.

Displaying V Variables

Displaying v variables containing "everyday" values -- not too large, not too small -- is quite similar to displaying n variables. Both the -show- and the -showt- commands may be used. The three-argument form of the -showt- command is the most straightforward v variable displaying tool. As in the case of n variables, the first argument tells what value is to be shown, the second argument tells how many character positions to supply before the decimal point, and the third, how many after the decimal point.

```
define  num=n1Ø,total=v2Ø,mean=v21,
        loc=n4Ø
        .
        .
        .
calc    num← 42
        total← 29Ø7.9
        loc← 261Ø
        .
        .
        .
calc    mean← total/num  $$ "mean" now equals 69.23571...
        .
        .
        .
at      loc
showt   mean,3,2         $$ displays   69.24  (one leading blank)
at      loc+1ØØ+3
showt   mean,3           $$ displays     69   (default, no third argument)
at      loc+2ØØ+2
showt   mean,1,2         $$ displays    ****  (error default, field too small
at      loc+3ØØ-6
write   Mean: <t,mean,3,2> $$  Mean: 69.24  (embedded form available too)
```

The great versatility of the -showt- command now becomes evident. With the three-argument form, a properly rounded value may be displayed with any format we choose. If the third argument is zero, or if only two arguments are given, a v variable will be displayed without a decimal point and thus appear to be an integer. This parallels the facility of making an n variable appear as though it were a floating point number by displaying it with a three-argument -showt-.

As a mere convenience, a one-argument default form exists. Like the case for n variables, a field of eight character positions is assumed. For v variables this field is partitioned to provide four positions before the

decimal, one position for the decimal point itself, and three positions for digits following the decimal point.

```
at      3ØØ8
showt   mean        $$ displays  69.236
at      31Ø8
showt   mean,4,3    $$ identical to preceeding -showt-
```

System Defined Variables

Certain kinds of variable information are used so frequently in so many different applications that the PLATO system keeps the information for each user and makes it available to the lesson author. These variables are called system defined variables or system reserved words. Their names and functions are already determined by the system, and thus you may not define them in your -define- command. But you may use them in your lesson just as you would use any student variable. There are more than 1ØØ system reserved words, but you need be concerned only with the most commonly used ones, such as "ntries" which contains the number of attempts the student has made to answer the current questions; "where" which contains the current coarse grid screen location ("wherex" and "wherey" contain the current fine grid x and y locations); "jcount", the number of characters in the student's answer.

Shown below are two versions of the same unit. One uses a student variable to count the number of attempts made by the student; the other uses the reserved word "ntries" for the same purpose.

```
define  attempt=n35
        .
        .
        .
arrow   121Ø
answer  water
calc    attempt=attempt+1
write   You got it in <s,attempt> tries!
wrong   (earth,air,fire)
calc    attempt=attempt+1
write   Try one of the other four elements of the ancients.
no
calc    attempt=attempt+1
write   It's H2O.
```

```
          .
          .
          .
arrow     121Ø
answer    water
write     You got it in <s,ntries> tries!
wrong     (earth,air,fire)
write     Try one of the other four elements of the ancients.
no
write     It's H2O.
```

Although the two units do the same thing, the one using "ntries" is simpler because the incrementing is taken care of automatically. System defined variables are like student variables in that each user has his own, and they may be used in any place (-calc- command, -show- command, or expression) that student variables can be used.

Variables with Numerical Responses

The introduction to the use of -ansv- and -wrongv- commands demonstrated applications in which explicit responses were anticipated, that is, a specific number appeared in the tag. We know now that a student variable or expression may be substituted in the tag.

```
define    term1=n1Ø,term2=n11,loc=n2Ø
          .
          .
          .
unit      setup
calc      term1← 12    $$ give values to two terms to
          term2← 5     $$ be multiplied
          loc← 1215    $$ screen location for problems
jump      multprob  $$ present multiplication problems to student
          .
          .
          .
unit      multprob  $$ displays a multiplication problem in
next      nextprob   $$ schoolbook fashion
at        loc
write       <t,term1,2>  $$ note that 2 spaces precede -showt-
          x <t,term2,2>  $$ mult. sign appears at left
at        loc+1ØØ
write     ____      $$ provide underscore
arrow     loc+2ØØ-1  $$ arrow appears to left of underscore
specs     noops     $$ "noops" means "no arithmetic operators allowed in respons
ansv      term1 x term2
write     Correct!
wrongv    term1 + term2  $$ check for common mistakes
write     Multiply, don't add!
endarrow
```

In unit "multprob", the correct answer depends on the current values of variables "term1" and "term2". This general utility unit, and others like it for addition, subtraction, and division problems, could be used to provide an elementary math student with an arithmetic practice drill. Such units are general in that the correct response and specific wrong responses are generated at the time the student is running the lesson. Only the general formula for the responses need be explicitly stated at the time the lesson is created by the author.

The -ansv- and -wrongv- commands permit the student to type an expression (such as 12x5). The expression is evaluated and the result is compared to the value of the tag to determine whether a match occurred. This attribute makes these commands too powerful to be used as they stand in a lesson testing arithmetic skills. The added judging specification

```
specs   noops
```

causes a "no" judgement to be rendered if the student's response contains an arithmetic operator (+, -, x, *, ÷, or /). In most applications one is testing the student's ability to solve a problem by piecing together an appropriate arithmetic expression. Having -ansv- and -wrongv- evaluate expression is the more general, default action. In the special application of testing arithmetic skills, this more general action should be disabled with the "noops" tag for -specs-.

Exercises

Write a multiplication drill which allows the student to do as many problems as he likes. The problems will be Nx1, Nx2, Nx3, etc, where N is any number you choose. You will need at least two defined variables, one for the multiplicand and one for the multiplier. The program will have two units, an initialization unit which sets one variable to the multiplicand you choose and sets the other to Ø, then jumps to the second unit. This second unit is the actual drill unit. In the drill unit, set NEXT to go to the drill unit again, and BACK to go to your index. Add one to the value of the second variable. Display the two variables as a multiplication problem (for example: write ⟨s,num1⟩ x ⟨s,num2⟩). Display the arrow. If the answer is correct, write "Press NEXT for another problem or BACK to return to the index". Check for the specific error of adding the two numbers. If that mistake is made, write an appropriate comment such as "Remember to multiply, not add!". For other errors, display some general comment such as "Try again". Remember that the -ansv- and -wrongv- commands can have defined variables and operators in their tags.

Chapter 6: Conditional Statements and Iterative Statements

One of the most useful ideas in computer programming is that of performing an operation only if a certain condition is met, or performing one of several different operations depending on a condition. Using information in student variables as the condition, most TUTOR commands can be written in a conditional format:

```
command  expression,case for negative,case for Ø,case for 1,
         case for 2,case for 3,...
```

When a conditional command is encountered, the expression in the tag is evaluated and rounded to the nearest integer. If the expression is negative, the argument in the negative position is used; if the expression is Ø, the argument in the zero position is used; if the expression equals 1, the argument in the one position is used, and so on. The last listed case is used not only for the value corresponding to its position, but also for all greater values. If no operation is to be done for a particular value, an x is used in that position in the tag. Below is a conditional -jump- command.

```
jump     n1,trig,geom,algebra,x
```

If n1 is negative, there will be a -jump- to unit "trig"; if n1 equals Ø,

Chapter 6: Conditional Statements and Iterative Statements

One of the most useful ideas in computer programming is that of performing an operation only if a certain condition is met, or performing one of several different operations depending on a condition. Using information in student variables as the condition, most TUTOR commands can be written in a conditional format:

```
command  expression,case for negative,case for Ø,case for 1,
         case for 2,case for 3,...
```

When a conditional command is encountered, the expression in the tag is evaluated and rounded to the nearest integer. If the expression is negative, the argument in the negative position is used; if the expression is Ø, the argument in the zero position is used; if the expression equals 1, the argument in the one position is used, and so on. The last listed case is used not only for the value corresponding to its position, but also for all greater values. If no operation is to be done for a particular value, an x is used in that position in the tag. Below is a conditional -jump- command.

```
jump     n1,trig,geom,algebra,x
```

If n1 is negative, there will be a -jump- to unit "trig"; if n1 equals Ø,

The two statements do the same thing, but the first requires counting the x's to determine for what value of "prob" unit "pictures" is done. The second statement shows at a glance that the unit is attached when "prob" equals 4.

Two or more logical expressions can be tested with the use of and and or, e.g.

```
do      (prob=4) $and$ (review=-1),extra,x
jump    (tries>2) $or$ (score<7Ø),morep,x
```

The -do- statement causes unit "extra" to be done only if both conditions are true. The -jump- is done if either condition is true. With and, both conditions must be true in order for the entire expression to be true. With or the true branch is taken if either condition is true.

Another conditional structure in TUTOR is the -if- command and its associates. The -if-, -elseif-, -else-, and -endif- commands allow the testing of logical conditions and the execution of specified commands based on the condition.

```
if      score>9Ø
.       write   Excellent!
elseif  score>8Ø
.       write   Good work!
elseif  score>7Ø
.       write   You completed the problems with a score of <s,score>.
else
.       write   Your score on the problems is <s,score>.  Let's review.
endif
```

The expressions used in the -if- structure must be logical expressions. The expression in the tag of the -if- is evaluated, and if it is true, the indented commands below the -if- are executed. If the expression is false, the indented commands are skipped and control passes to the -elseif-. That expression is evaluated, and if true, the indented commands are executed. If the expression is false, each -elseif- in turn is treated, and if all are false, control passes to the -else-. The commands below the -else- are executed if none of the previous conditions holds true. The -endif- serves as a marker for the end of the structure.

Statements below -if-, -elseif-, and -else- must be indented: the line must begin with a period followed by 7 spaces. To type the period and 7 spaces, press MICRO-period or press period and then TAB.

Iterative Statements

We have seen how to attach units with -do-. If the unit is to be executed several times consecutively, the iterative form of the -do- command is used. The iterative form of -do- is

```
do      unitname,index variable← starting value,ending value,increment
```

A typical iterative -do- statement might be

```
do      counter,n1← 1,1Ø,2
```

This means "do unit counter while n1 goes from 1 to 1Ø in steps of 2." The increment is optional, and if left out, a 1 is assumed. The statements below execute unit "box" three times.

```
unit    drawit
calc    corner← 81Ø
do      box,count← 1,3,1
at      4Ø5
write   See the boxes!
*
unit    box
box     corner;(corner-2ØØ+2Ø)
calc    corner← corner+4ØØ
```

The -calc- command sets "corner" to 81Ø. The variable "count" is set to 1, and unit "box" is executed. After "box" is executed, "count" is incremented by 1. If the resulting value is not greater than the ending value (3 in this case), unit "box" is executed again. The value of the expression (count+increment) and the ending value are again compared. This continues until (count+increment) is greater than the ending value. Unit "box" is executed three times, drawing a box at three different locations on the panel. The action of units "drawit" and "box" are shown in the flow chart in figure 6.1.

It is often convenient to choose starting and ending values which can be used within the loop. We could rewrite units "drawit" and "box" in this manner:

```
unit    drawit
do      box,corner← 81Ø,161Ø,4ØØ
at      4Ø5
write   See the boxes!
*
unit    box
box     corner;(corner-2ØØ+2Ø)
```

The value of "corner" is set to 81Ø and unit "box" is executed. On the next iteration the value in "corner" is incremented by 4ØØ, giving 121Ø. Unit "box" is executed with the new value. Variable "corner" is again incremented by 4ØØ and unit "box" is executed with the value 161Ø in "corner".

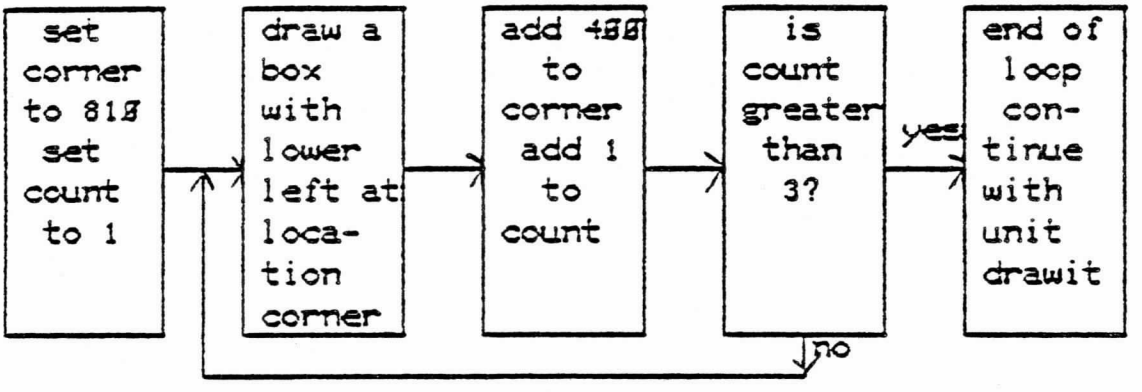


Figure 6.1 Flow chart of iterative -do-

Sometimes it is advantageous to put repeated code within the current unit rather than in an attached unit. For example, lengthy -calc- statements will execute much faster if written on consecutive lines than if interspersed among other code or dispersed among attached units. The -doto- command, which repeatedly executes statements within the current unit. A statement label marks the end of the code to be repeatedly executed. These units present exactly the same display as those above, but the variable "corner" serves as both the index variable and the location of the lower left corner of the drawing. Figure 6.2 shows a flowchart of the modified units.

```
unit    draw3
doto    3box,corner← 81Ø,161Ø,4ØØ
box     corner:(corner-2ØØ+2Ø)
3box
at      4Ø5
write   See the boxes!
```

The statements between the -doto- and the statement label -3box- are executed repeatedly until the index variable plus the increment is greater than the final value. After the last iteration, the commands below the statement label are done. A statement label must begin with a number and must be 7 or fewer characters long. Although the examples of -doto- in this chapter have all used constants for the starting and ending values of the loop, variables may be used.

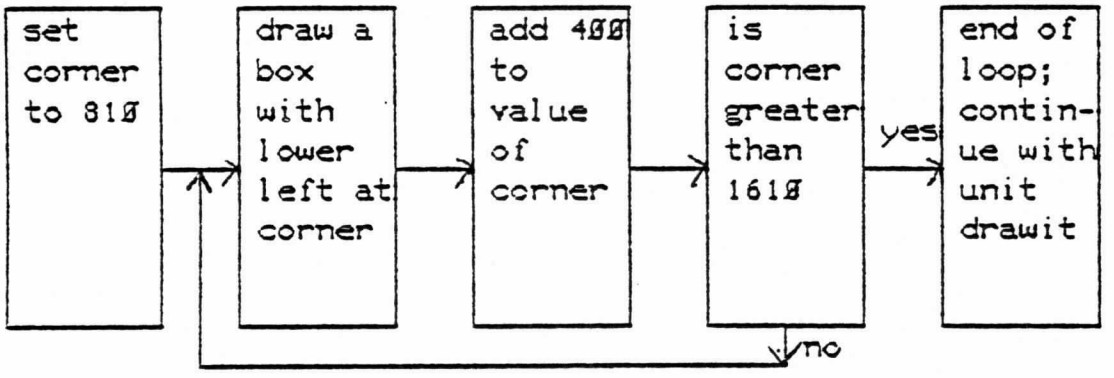


Figure 6.2 Flow chart of modified unit "drawit"

The -branch- command is also used to pass control of execution within a unit. Like -doto-, it refers to a statement label within the unit. It causes execution to skip to the command below the statement label referred to in the tag. The -branch- command may be used conditionally:

```
unit    endprob
if      score>7Ø
.       jump    alldone
else
.        calc   revu← -1
.        jump   probs
endif
*
unit    probs
branch  revu,x,lskip  $$ skip -write- if not reviewing
at      3Ø5
write   Review
lskip
at      1215
write   .....   $$ first problem
        .
        .
        .
```

The variable "revu" is set to -1 only if the student is reviewing the problems. If so, the word "Review" appears in the upper left corner of the screen. For students who are not reviewing, the commands between the -branch- and the statement label "lskip" are not done.

Indirect Referencing of Variables

So far in our use of variables, we have referred to them directly, by name, and in -define- by number. The primitive names may have their number enclosed in parentheses: n(42) is exactly equivalent to n42. The feature is convenient when using iterative operations because it allows executing the same operation on several consecutive variables. The TUTOR code below uses indirect referencing to display the contents of n1 through n1Ø.

```
doto    lshow,count← 1,1Ø
at      2Ø5+(count*1ØØ)
show    n(count)
lshow
```

On the first iteration, the defined variable "count" has the value 1, so the statement -show n(count)- is equivalent to -show n(1)- or -show n1-. On the second iteration, "count" is equal to 2, so the tag n(count) is equivalent to n2; etc. By using a name instead of a number, we have referred to the variable indirectly.

Indirect references may be set up in the -define- statement:

```
define  choice(xx)=n(xx)
```

The "xx" is a so-called dummy argument. It is not defined. When the variable "choice" is used in the program, the value given in parentheses is substituted for the "xx". For example, the statement -calc choice(4)← 17- stores the value 17 in n4. With a -define- such as

```
define  choice(xx)=n(1Ø+xx)
```

"choice(1)" refers to variable 11, "choice(2)" refers to variable 12, and so forth.

The code below shows a -doto- loop using a defined function.

```
define  rate(xx)=n(xx+5Ø)
        index=n15Ø
        .
        .
        .
doto    1Øshow,index← 1,1Ø
at      2Ø5+(index*1ØØ)
show    rate(index)
1Øshow
```

On the first iteration the contents of n51 are shown, on the second, the contents of n52 are shown, and so on through n6Ø.

These features can be used to show on an index page which sections the student has completed. A variable can be set in the last unit of each section. In the index unit, the variables can be examined, and for each variable which is set (indicating that the student passed through the final unit) an asterisk can be written on the screen. The units below show how this might be done.

```
define  done(zz)=n(3Ø+zz)  $$ n31, 32, 33 used as flags
        lindex=n34          $$ n34 is index for loops
calc    done(1)← done(2)← done(3)← Ø  $$ set 3 vars to Ø
*
unit    index
at      91Ø
write   Choose a topic:
at      1415
write   a  Pulse Rate

        b  Blood Pressure

        c  EKG
at      2Ø1Ø
write   Topics you have completed are marked with *
doto    3parts,lindex← 1,3
at      1Ø2Ø+(2ØØxlindex)  $$ move down 2 lines
writec  done(lindex),,,*   $$ write * if value = 1
3parts
*
*  more TUTOR code
*
unit    endpr  $$ last unit of part a
calc    done(1)← 1
*  more TUTOR code
unit    endbp  $$ last unit of part b
calc    done(2)← 1
*  more TUTOR code
unit    endekg  $$ last unit of part c
calc    done(3)← 1
*  more TUTOR code
```

The values of "done(1)", "done(2)", and "done(3)" are set in the last unit of the topic. In the index unit, those variables are examined. If they are zero, nothing is written. If the value of any one of them is 1, an asterisk is written on the screen beside the corresponding topic.

Exercises

Modify the multiplication drill from Exercise 5 so that the user returns to the index after 8 problems rather than at his discretion. This can be done by using a variable to count the number of times the problems unit is executed and including a conditional -jump- which returns to the index when the variable reaches a value greater than 8. Also add conditional statements to show the student the correct answer after 3 incorrect attempts.

Both these modifications can be made using -if- structures instead of commands in the conditional form. You may wish to try both approaches.

Optional Exercise

Add the necessary commands so that the index page of your lesson indicates which sections (geometry, TUTOR, multiplication) the user has completed. You may indicate completion by an asterisk or some other symbol, or by writing the word "completed" next to the topic.

Chapter 7: More Judging Capabilities

We have seen how to use -answer-, -wrong-, -ansv-, -wrongv-, and -no- to judge student responses. In this chapter we will consider judging and setting a variable in one step with the -match- command, storing the response in variables with -store- and -storea-, accepting responses unconditionally with the -ok- command, and forcing immediate judgment with the -long- and -force- commands. We will also examine features of the -specs- command.

The maximum number of characters normally permitted in a student response is 15Ø. This default can be altered by the -long- command. The tag is the number of characters (from 1 to 3ØØ) to be allowed. The -long- command should follow the -arrow- and precede the first judging command. When the student's response reaches the maximum number of characters, no more characters can be typed. The student must press NEXT to initiate judging or else erase some characters. If desired, judging can be initiated as soon as the length limit is reached, without the student's pressing NEXT. This is accomplished with -force long-. When -long 1- is used, immediate judgment is automatic; no -force long- is necessary.

The -match- command searches the student's response for one of a number of listed words and sets a variable indicating which one was found. A judgment is always made by the -match- command; judging state ends and the regular commands which follow are executed. The syntax of -match- is:

match variable,item0,(item1,synonym1),(item2,synonym2),item3,...itemn

For example:

```
match   n1,a,b,c,d,e
```

If the student responds "a" n1 is set to Ø, if he responds "b" n1 is set to 1, and so forth. If the response matches none of the listed items, n1 is set to -1 and a "no" judgment is given. The code below shows the use of -long 1- and -match- in an index unit.

```
unit    index
at      7Ø5
write   Press the letter of the topic
        you want to study.
at      91Ø
write   a. Verbs

        b. Nouns

        c. Pronouns

        d. Adjectives & Adverbs
inhibit arrow  $$ so arrow won't show on screen
arrow   1Ø5
long    1  $$ student needn't press NEXT
match   part,a,b,c,d
jump    part,x,verb,noun,pron,ad,x
write   Choose a listed letter, please.
```

As soon as the student presses a letter, judging begins because of the -long 1- statement. If his choice matches an item in the tag of the -match- a -jump- to the appropriate unit occurs. If not, the -write- below the -jump- is done.

A student's input may be stored in variables. The -store- command evaluates a numerical response and stores the result in the variable named in the tag.

```
store   v16  $$ stores result of numerical response in variable 16
```

The -store- command is a judging command. Other judging commands which we have studied cause judging state to end when they are matched; -store-, however, ends judge state only if the response cannot be evaluated. Things which -store- cannot evaluate are non-numerical input and invalid

expressions (e.g. "2x12+"). This is a convenient feature because it allows storing the student's response, continuing judging, and comparing the response to the tag of another judging command. The stored response can be displayed. In the TUTOR code below, the student's reponse is stored in v1Ø.

```
arrow   1218
store   v1Ø
write   Enter a number or expression.
ansv    v12
write   Good work.
no
write   The correct answer is ⟨s,v12⟩.  You said
        ⟨s,v1Ø⟩.  Try again.
```

The -write- below the -store-, since it is a regular command, is done only if the student's expression is invalid. For valid expressions, judging state continues. The -write- is skipped and the response is compared to the tag of the -ansv-. The -write- below the -no- shows the student his own response as stored by the -store- command.

When a -store- command is used with no other judging commands, it is necessary to end judge state. The -ok- command ends judge state unconditionally. It is just like the -no- command except that it gives an "ok" judgment rather than a "no". When in regular state after judging state has ended, it is possible to switch back to judging state with the -judge- command. Some tags of -judge- are "continue", which switches back to judge state, and "ok" and "no" which return a judgment of "ok" or "no" respectively and remain in regular state. Since -judge- is used after a judgment has been made, it is of course a regular, not a judging, command. A conditional use of -judge- after a -store- command allows for checking that the student's number is within an allowed range. Illustrated below is the use of -store-, -judge-, and the system reserved word "judged" which has the value -1 if the judgment was "ok", Ø if a -wrong- command was matched, and 1 if a -no- was matched or a -judge no- was matched.
The student's response is stored in the defined variable "topic". The -ok- ends judging state so that -judge-, a regular command, can be used to alter the judgment if the number is outside the permitted range. If the number is out of range, the judgment is switched to "no"; if not, no change is made in the judgment. The -writec- based on the value of the system reserved word "judged" displays a message if the judgment was "no".

```
unit    index
at      7Ø5
write   Press the number of the topic
        you want to study.
at      91Ø
write   1.  Verbs

        2.  Nouns

        3.  Pronouns

        4.  Adjectives & Adverbs
inhibit arrow  $$ so arrow won't show on screen
arrow   1Ø5
long    1  $$ student needn't press NEXT
store   topic
write   Choose a listed topic.
ok
judge   (topic < 1) $or$ (topic > 4),no,x
jump    topic,x,x,verb,noun,pron,ad,x
writec  judged,..Choose a listed topic.
```

Up to now we have used variables only for storing numbers, but alphabetic characters can be stored as well. Alphabetic information can be stored in variables with -storea-, a judging command. The -showa- command displays the contents of a variable in alphanumeric form.

```
define   stuname=n1
         .
         .
unit     name
at       14Ø8
write    What do you want PLATO to call you?
arrow    161Ø
storea   stuname  $$ store answer as characters
ok                $$ end judge state
write    Fine,
showa    stuname  $$ display contents of variable
```

The -storea- command does not end judge state, so it must be followed by the -ok- command. The -showa- can be embedded, just like -show- and -showt-. The unit above could be rewritten like this:

```
unit     name
at       14Ø8
write    What do you want PLATO to call you?
arrow    161Ø
storea   stuname  $$ store answer as characters
ok                $$ end judge state
write    Fine, <a,stuname>
```

N variables, rather than v variables, should be used for storing alphanumeric information. Up to ten characters can be stored in each TUTOR variable. We can think of each TUTOR variable as having 1Ø "places" for characters, like this: ☐☐☐☐☐☐☐☐☐☐. The -storea- command places the characters in the variable starting at the left. Thus the unit below

```
unit    name
at      14Ø8
write   What do you want PLATO to call you?
arrow   161Ø
storea  stuname  $$ store answer as characters
ok               $$ end judge state
write   Fine, <a,stuname>
```

places the student's name in "stuname" like this:

b	i	l	l						

A -showa- of the variable produces "bill". A capital letter takes up two characters, one for the shift code and one for the letter itself. So if the student uses a capital, the variable looks like this:

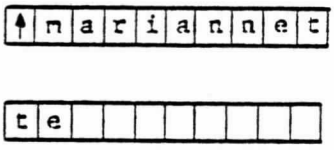
↑	b	i	l	l					

A -showa- of the variable shows "Bill".

Both -storea- and -showa- have a default length of 1Ø. That is, unless you specify otherwise, -storea- stores up to 1Ø characters and -showa- displays up to 1Ø. If you specify a longer length, the next consecutive variable is used for the characters beyond the tenth.

```
define  stuname=n1
* n2 left open to allow for a name up to 2Ø chars long
        index=n3
        .
        .
unit    name
at      14Ø8
write   What do you want PLATO to call you?
arrow   161Ø
storea  stuname,2Ø  $$ allow up to 2Ø chars
ok
write   Fine, <a,stuname,2Ø>  $$ display contents
* of "stuname" & of following variable
```

A student response like "Mariannette" will be stored like this:



The first 1Ø characters are stored in n1 and the remaining characters in n2.

We have seen that the -specs- command sets specifications for interpreting the student's reponse. The tag of "bumpshift" causes an "ok" judgment for responses which match the -answer- tag in all respects except capitalization. Other tags set other specifications. A single -specs- command may have several tags:

```
specs   okspell,bumpshift,noorder
```

indicating that spelling, capitalization, and order of words are not important. A response which matches a subsequent author answer in all respects but these three is judged "ok". For example, the commands

```
specs   okspell,bumpshift,noorder
answer  <george,general> washington was <the> first president
```

cause an "ok" judgment for responses such as "The first president was Washington" or "George Washington was first President".

The tags of -specs- can be divided into five functional categories: those which affect spelling and punctuation, those affecting extra words and word order, those dealing with numerical responses or parts of responses, those which affect markup of the response and automatic feedback, and those affecting letter-number interpretation. Complete details of all these tags can be found in AIDS.

The -specs- command serves another function besides setting judging specifications. It is an important marker in the execution of TUTOR. After any judgment is made, if there is a -specs- command, control returns to the -specs- and any regular commands between the -specs- and the next judging command are executed. This is done whether the judgment was "ok" or "no", so it is a convenient place to put commands which should be done no matter

which judging command was matched. The TUTOR code below shows the use of -specs- as a marker.

```
          .
          .
          .
arrow     13Ø7
specs
at        151Ø
writec    spell,,Underlining indicates a misspelled word.
answer    (epidemiology,epidemiological,epidemic) <concerns,matters,interests>
do        epidemic
answer    (pathology,pathological) <concerns,matters,interests>
do        path
no
write     Press DATA for some review material.
```

The -writec- based on the system reserved word "spell" is done if the student misspells a word in either of the -answer- tags. The marker function is fulfilled by -specs- whether or not it has a tag. If there is more than one -specs- associated with an -arrow-, the last one serves as the marker.

Exercises

1. Add a unit at the beginning of your program to ask the student what he wants to be called. Store the name, and display it at that point, and if you wish, at any other appropriate points in the lesson.

2. Add a -long 1- and to your index unit. Change the index unit so that, rather than -answer- commands followed by -jump- commands, you use either -match- and a conditional -jump- or -store- and a conditional -jump-.

3. Add a -writec- to the multiplication drill to display "Give a value, not an expression" if the student uses mathematical operators in his response. The -writec- will be based on the system reserved word "opcnt". To avoid repeating the -writec- after every judging command, place it below the -specs- command.

Chapter 8: Special Branching

We have used the NEXT, BACK, SHIFT-NEXT, and SHIFT-BACK keys extensively to move to a different main unit. The HELP, LAB, DATA, SHIFT-HELP, SHIFT-LAB, and SHIFT-DATA keys, referred to collectively as help-type keys, are used to provide help to the student. They may be used in two different ways: to provide help in the current main unit or to provide help in a new main unit or series of main units. To provide help within the current main unit, use the -helpop-, -labop-, -dataop-, -helplop-, -lablop-, and -datalop- commands. The -helpop- command activates the HELP key, and the -helplop- command activates the SHIFT-HELP key. The same is true for the other "op" commands: the l indicates the shifted key. The "o" and "p" in the name stand for "help on the same page." The tag of the -helpop- (etc.) command is the name of the unit to be attached if HELP (etc.) is pressed. The statement

```
labop   hints
```

will cause unit "hints" to be executed if LAB is pressed. After unit "hints" is executed, the student continues in the main unit at the point at which he pressed the key. These commands work just like -do- except that the unit is attached under control of a student keypress.

The use of help on the same page is illustrated below.

```
unit    verbs
helpop  endings
at      302
write   Fill in the blank with the correct form of the verb "finir":
        Press HELP for help with the endings.
doto    9verbs,person← 1,9
at      608+(200*person)
writec  person,,,On,Tu,Elles,Nous,Vous,Il,Je,Elle,Ils
draw    608+(200*person)+6;608+(200*person)+19
at      where+2
write   les leçons à trois heures.
arrow   608+(200*person)+5
answerc person;;;finit;finis;finissent;finissons;finissez;
        finit;finis;finit;finissent
endarrow
9verbs
*
unit    endings
at      2802
write   Endings for ir verbs:
             is      issons
             is      issez
             it      issent
```

The student may press HELP at any time, and be shown the -write- statement in unit "endings". After the display is shown, the student continues at his current place in unit "verbs".

The -helpop-, etc., commands provide help on the current display, and thus are used when the information to be shown is short and immediately relevant to the details of the current main unit. Sometimes it is useful to give longer and more general help. The commands -help-, -lab-, -data-, -help1-, -lab1-, and -data1- provide help in a new main unit. If one of these commands is present, when the student presses the corresponding key a help sequence is initiated. A help sequence has several automatic features. The student is branched to the unit named in the tag. The help sequence may be one or more units long. When the help sequence is ended, the student is returned to the beginning of the main unit from which he pressed the help-type key. The main unit from which the help-type key was pressed is called the base unit. The system automatically remembers the base unit, so it is possible to use the same help sequence in several different places and yet return to the correct base unit at the end of the help sequence. The end of the help sequence is marked by an -end help- statement or simply an -end- with no tag. Pressing NEXT in a unit containing an -end- statement or pressing BACK or SHIFT-BACK in any unit in a help sequence returns the

student to the base unit. The tag of the -help- (or -data-, etc.) command is the name of the unit to proceed to if HELP (or DATA, etc.) is pressed. The units below illustrate a help seqence.

```
unit    text1
next    text2
back    intro
help    h1
        .
        .
        .
unit    text2
next    text3
back    text1
help    h1
        .
        .
        .
unit    text3
next    text4
back    text2
help    h1
        .
        .
        .
unit    h1
next    h2
        .
        .
        .
unit    h2
back    h1
        .
        .
        .
end     help
```

The HELP key is active in units "text1", "text2", and "text3". Pressing HELP takes the student to unit "h1". NEXT from unit "h1" goes to unit "h2". While in unit "h2" the student may review unit "h1" by pressing BACK. Pressing BACK from unit "h1" returns the student to the unit from which he pressed the HELP key. Pressing SHIFT-BACK from either of the help units returns the student to his base unit. And because unit "h2" contains the -end help- statement, pressing NEXT from unit "h2" also returns the student to the base unit. A flow chart of these units (figure 8.1) shows what unit the student sees when he presses various keys.

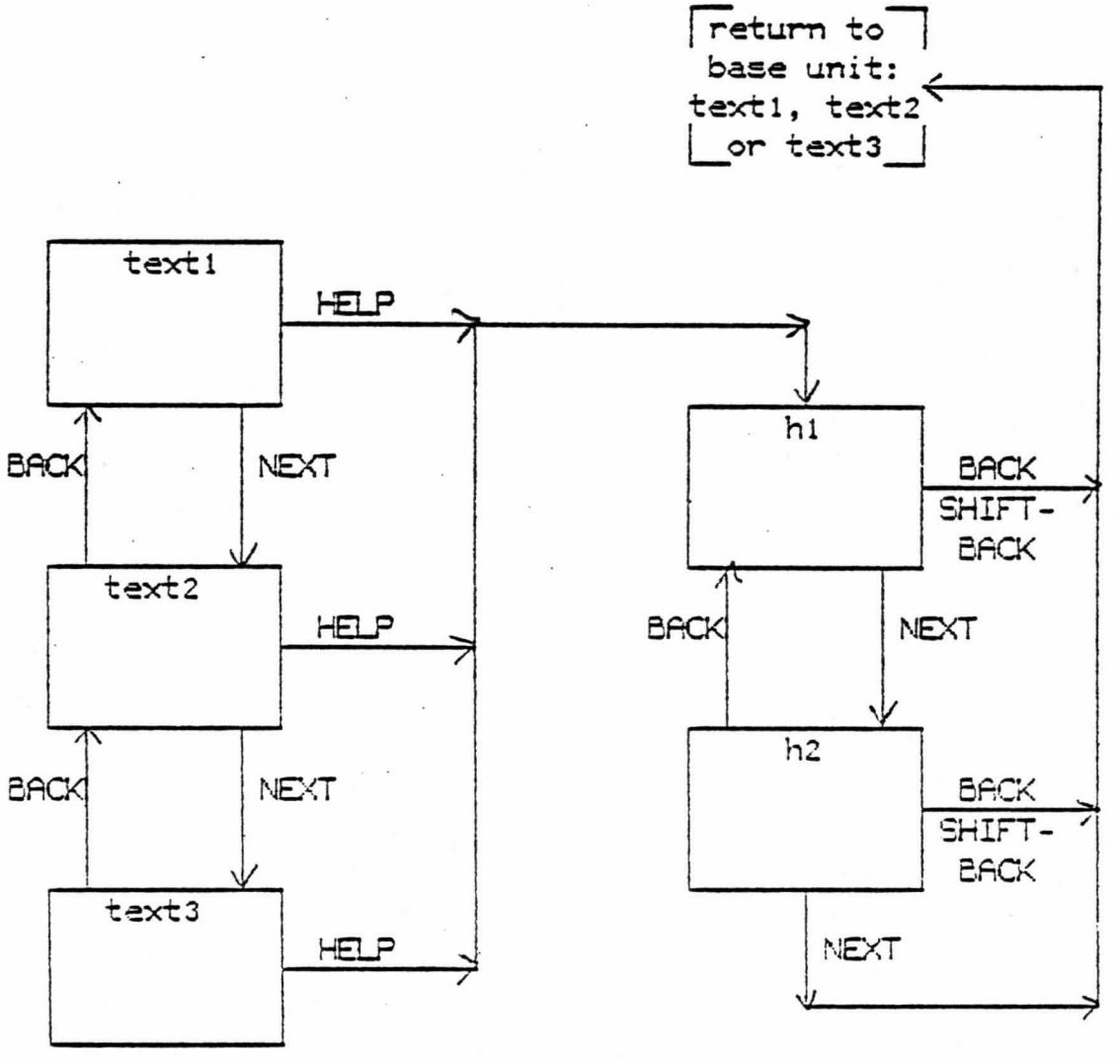


Figure 8.1 Flow Chart of a HELP Sequence

If the student presses BACK or SHIFT-BACK while in a help sequence, he is returned to the base unit. However, if there is a -back- or -back1- command in the help sequence and the student presses the corresponding key, he is branched to the unit named in the tag of the -back- or -back1- command. In other words, the execution of a -back- or -back1- command takes precedence over the default function of BACK and SHIFT-BACK in help sequences.

A student always has a current main unit. That main unit may have a unit attached to it by -do-. But there is a base unit only when the student is in a help sequence. When the help sequence is ended and the student returns to the unit from which he entered the help sequence, he no longer has a base unit. When the student is in a help sequence, the system reserved word "baseu" is set to the name of the base unit. When the help sequence is completed "baseu" is set to Ø. Note that only the -help-, -lab-, -data-, -help1-, -lab1-, and -data1- commands set "baseu". The -helpop-, etc., commands do not.

The TERM key is a help-type key, but its use and behavior are somewhat different from the other help-type keys. The other help-type keys provide student-initiated branching from a specific unit to a specific unit. The -term- command provides student-initiated branching from <u>any</u> unit in the lesson to a specific unit. Its tag is not a unit name; it is a word that the student will type. The -term- command is placed in the unit the student is to be sent to.

```
unit    one
next    two
          .
          .
          .
at      3ØØ5
write   For a list of definitions, press TERM and type "words".
unit    two
          .
          .
          .
*
unit    defs
term    words
          .
          .
          .
write   Press BACK to return.
end     help
```

In any unit in the lesson, when the student presses TERM, the phrase "What term?" and an arrow appear at the bottom of the panel. If the student types "words" and presses NEXT, he is taken to unit "defs". From unit "defs", pressing BACK or SHIFT-BACK returns him to the main unit from which he pressed the TERM key. Pressing NEXT has the same effect, since unit "defs" contains an -end- command. The TERM key is active anywhere in a lesson. If the word typed by the student exactly matches the tag of a -term- command, he is taken to the unit containing that -term- statement. If his word does not match the tag of any -term-, his input and the "What term?" message are erased, and the TERM keypress is ignored.

The -termop- command works like the -term- command, except that it does not initiate a help sequence. As with the other "op" commands, the current main unit is not changed and at the end of the termop unit execution of the current main unit continues.

Altering the Base Unit

The base unit may be altered by the lesson, if desired, with a -base- command. The -base- command with no tag clears the base pointer so the student is no longer in a help sequence. That is, a -base- command with a blank tag changes a help sequence into a regular lesson sequence. This is desirable when TERM is used to go to an index of topics, for example. In such a case the base pointer should be cleared so that the flow of control is unhampered.

```
unit    topics
term    index
base
at      803
write   Press the number of the topic you want to study.
        .
        .
        .
```

Failing to clear the base pointer in a case like this could cause the student to leave unit "topics" with the base pointer set, proceed through a section of the lesson, enter another HELP sequence, and at its end be returned to some unit relevant to the section he was in before he entered the current help sequence.

It is sometimes useful to return a student to a unit preceding the one from which he entered a help sequence. In the code below, the LAB key is active in two units. However, the student returns to unit "meteor1" after the help sequence, no matter which unit he was in when he pressed LAB.

```
unit    meteor1
lab     mhelp
next    meteor2
        .
        .
        .
unit    meteor2
lab     mhelp
next    meteor3
        .
        .
        .
unit    mhelp
base    meteor1
        .
        .
        .
end     help
```

Pressing LAB from either "meteor1" or "meteor2" takes the student to unit "mhelp". The -base- command in unit "mhelp" sets the base pointer to "meteor1" so that at the end of unit "mhelp" the student returns to unit "meteor1".

In our discussions of TUTOR sequencing so far, we have assumed implicitly that the student begins and finishes a lesson in one session. Very often this is not the case. To ensure that the student does not repeat sections of a lesson unnecessarily, the -restart- command is used. The form of the command is:

```
restart   unitname
restart             $$ blank tag means current main unit
```

When a -restart- is encountered, a pointer is set to the current main unit (if blank tag) or to the unit named. If the student signs off and later signs on again, he begins work in the main unit indicated by the restart pointer. His 150 student variables will have been saved when he signed off

and restored when he returns, so everything we know about his performance in the lesson is just as it would have been had he worked without interruption. Several -restart- commands should be placed at logical points in the lesson.

Any commands placed before the first -unit- command in a lesson constitute an initial entry unit, or ieu. The -define- command and commands that load special characters into the terminal should be placed in the ieu. Commands in the ieu are executed every time the user enters the lesson, even if he does not start at the beginning.

A lesson is ended when the student completes the last physical unit in a lesson, but a much better way to mark the lesson's end is with the -end lesson- or -lesson complete- statement. Either of these can be placed in the last logical unit of the lesson. Explicit ending with -end lesson- or -lesson complete- is preferable to default ending, for the same reasons that explicit flow of control with -next- is preferable to default flow of control.

A lesson may need to set certain variables or perform other operations when the student leaves the lesson without completing it. These can be done in a "finish unit", which is a unit done when the student leaves by pressing SHIFT-STOP. (It is not done when the student leaves by encountering an -end lesson- or -lesson complete-.) A finish unit is declared in a -finish- command. Its tag is the name of the unit to be executed when SHIFT- STOP is pressed. A finish unit cannot have any commands which send output to the screen, such as -write- or -draw-. It should be used mainly for calculations.

If a unit needs to be attached to every main unit, or at every arrow, the unit can be declared with an -imain- or -iarrow- command. The unit named in the tag is then done in every main unit (if -imain- command) or at every arrow (if -iarrow- command). An imain unit is frequently used to activate some branching and write on the screen which keys are available. An iarrow unit is a convenient place to put commands like -long- or -force- which should be active at every arrow. The feature can be turned off by executing an -imain- or -iarrow- with a blank tag, so they can be used only in particular sections of the lesson. The code below shows a portion of a lesson using -imain-.

```
* icu containing defines
          .
          .
unit    expo1    $$ several units of expository
next    expo2    $$ material
          .
          .
          .
unit    expo5    $$ last unit of expository material
next    check1
          .
          .
          .
unit    check1   $$ several units checking student's
*                $$ understanding of expository material
imain    keys    $$ unit "keys" will be done in every main unit
next    check2
          .
          .
          .
unit     check5  $$ last checking unit
imain            $$ turn off imain, disable unit "keys"
next     summary
          .
          .
          .
unit     keys
data     aux    $$ enable DATA key
help     tenses    $$ enable HELP key
lab      st1    $$ enable LAB key
at       3241
write    HELP LAB DATA available
```

The -imain- unit, unit "keys", enables the HELP, LAB, and DATA keys, and writes on the screen. This unit is attached for every main unit between unit "check1" and "check5", thus being available throughout that section of the lesson. The -imain- with no tag in unit "check5" disables the imain action for subsequent units.

Exercises

1. Make sure all definitions of variables are located under a single -define- command in the ieu.

2. Provide a way for the student to return to the index at any time. This could be done with a -term- command in the index unit or a -backl- command in an imain unit. If you use -term-, be sure to clear the base unit upon return to the index.

3. Add a "helpop" unit (or dataop, labop, etc.) to the first question unit on TUTOR commands. This unit should display a short list of possible answers to the question.

4. Add a help sequence (-help-, -lab-, -data-, etc. command) available from the second question unit on TUTOR commands. In the help sequence, give a definition of the term 'main unit' which the student can use to answer the question.

Index