DOCUMENT RESUME

ED 124 149 IR 003 537

AUTHOR Sherwood, Bruce Arne
TITLE The TUTOR Language.
INSTITUTION Illinois Univ., Urbana. Computer-Based Education Lab.
PUB DATE Jun 74
NOTE 225p.; Charts may be marginally legible due to print size
AVAILABLE FROM PLATO Publications, Computer-based Education Research Lab, 252 Engineering Research Lab, University of Illinois, Urbana, Illinois 61801 ($2.30, prepayment required)

EDRS PRICE MF-$0.83 HC-$11.37 Plus Postage.
DESCRIPTORS Autoinstructional Aids; *Computer Assisted Instruction; Computer Graphics; Computer Oriented Programs; Computer Programs; *Curriculum Development; Educational Technology; Higher Education; Individualized Instruction; Input Output Devices; Instructional Media; Manuals; Programed Instruction; Programed Tutoring; *Programing Languages
IDENTIFIERS *PLATO IV; Programmed Logic for Automatic Teaching Operations; *TUTOR

ABSTRACT

This document explains the TUTOR language which is used by teachers to create or author lesson materials on the PLATO IV computer-based education system. After an introductory section, the second section explains how to display text and line drawings to students. The third section introduces subroutines, and it is followed by explanations of calculations in TUTOR. The fifth section discusses in detail how to build interconnections and sequences into a lesson. Conditional commands are explained in section six and section seven explains how to judge student responses. Additional display features are described in section eight and additonal calculation topics are given in section nine. Common variables are listed in section ten, and the document concludes with a miscellaneous section. Sources of further information, a list of TUTOR commands, and built-in calculation functions are provided in the appendix. (CH)

The TUTOR Language

Bruce Arne Sherwood

Computer-based Education Research Laboratory
and Department of Physics
University of Illinois, Urbana

First Printing June 1974
Second Printing June 1975

Minor corrections have been made. Footnotes indicate areas of TUTOR where new features have been implemented. Full descriptions of these new features are available through on-line PLATO documentation.

Preface

The PLATO IV computer-based education system was developed in the Computer-based Education Research Laboratory (CERL) of the University of Illinois, Urbana. PLATO IV is the result of fifteen years of research and development effort led by Donald Bitzer, director of CERL. The system presently links nine hundred graphical-display terminals to a large computer in Urbana. Some of these terminals are located as far away as San Diego, Toronto, and Washington, D.C. Students are tutored individually at terminals by interacting with PLATO lesson materials created by teachers. There are over two thousand hours of PLATO lessons available at all terminals. These lessons span a wide range of subject areas and are used by students in elementary schools, community colleges, military training bases, universities, and commercial training programs. Authors of lesson materials are teachers who use the TUTOR language to tell PLATO how to interact with students on an individual basis. This book explains the TUTOR language in detail and is intended to help authors write quality lesson materials.

In 1967 Paul Tenczar, then a graduate student in zoology, concluded that existing methods of creating computer-based lesson material on the earlier PLATO III system were unnecessarily difficult. As a result he originated the TUTOR language. There followed a rapid increase in the number of authors and in the number and sophistication of the lessons they wrote. This active author community in turn spurred the continual development and refinement of TUTOR by requesting additional needed features. In 1970 CERL began implementing the PLATO IV system, which afforded a rare opportunity to take stock of the evolution of TUTOR up to that point and make a fresh start. Many useful simplifications were made, and many important features were added. The growth of PLATO IV into a continental network brought together an ever-wider spectrum of authors through the rich interpersonal communications facilities available on PLATO, and the suggestions and criticisms from these authors contributed to the present form of the TUTOR language. Also of great importance has been the large number of students who have used PLATO lessons and whose experiences have influenced the development of TUTOR to meet their needs. The TUTOR language described in this book is, therefore, based on heavy use-testing.

In the earliest phase Paul Tenczar and Richard Blomme were mainly responsible for TUTOR development. Since then, many people have been involved, some as full-time CERL staff members and some as high-school, undergraduate, or graduate students. It is impossible to acknowledge adequately the various contributions, and difficult even to list all those who have played a major role, but an attempt should be made. Paul Tenczar is head of TUTOR development. Full-time people have included David Andersen, Richard Blomme, John Carstedt (CDC), Ruth Chabay, Christopher Fugitt, Don Lee, Robert Rader, Bruce Sherwood, and Michael Walker.

They have been assisted by James Parry, Masako Secrest, and Donald Shirer, and by Doug Brown, David Frankel, Steven Freyder, Sherwin Gooch, David Kopf, Kim Mast, Phil Mast, Marshall Midden, Louis Steinberg, Larry White, and David Woolley.

All of these people have been involved mainly with "software", the programming of the PLATO computer in such a way as to permit authors and students to write and use computer-based lessons. Of equal importance to the technical success of PLATO are the CERL engineers, scientists, and technicians who invented, designed, and implemented the unique terminals and telecommunications devices ("hardware") which form the PLATO educational network. Many attractive features of the TUTOR language are closely related to the unusual characteristics of these devices. CERL personnel who have been heavily involved in hardware development include Donald Bitzer, Jack Stifle, Fred Ebeling, Michael Johnson, Roger Johnson, Frank Propst, Dominic Skaperdas, Gene Slottow, and Paul Tucker.

The PLATO computer is a Control Data Corporation Cyber computer, whose Extended Core Storage provides the high-speed swapping memory essential to the implementation of PLATO and TUTOR.

The latter part of chapter I is adapted from a PLATO III document, "The TUTOR manual", by R.A. Avner and P. Tenczar.

I thank Elaine Avner for editorial assistance, Sheila Knisley for typing, and Stanley Smith for photographic work. I appreciate the encouragement William Golden gave me to finish the task.

CONTENTS

I. Introduction

This book describes in detail the TUTOR language, which is used by teachers to create lesson materials on the PLATO IV computer-based education system. Teachers use the TUTOR language to express to the PLATO computer how PLATO should interact with individual students. Students and teachers interact with PLATO through terminals each of which includes a plasma-panel display screen and a typewriter keyboard, as shown. Using TUTOR, an author of a computer-based lesson can tell PLATO how to display text, line drawings, and animations on the student's screen. The author can ask PLATO to calculate for the student, to offer the student various sequencing options, and to analyze student responses.

It is hoped that you have already studied the textbook "Introduction to TUTOR" by J. Ghesquiere, C. Davis, and C. Thompson, and the associated PLATO lessons. These materials are designed to teach you not only basic aspects of TUTOR but also how to create and test your own lessons on the PLATO system. The present book, "The TUTOR Language", does not attempt to describe the latter aspects, such as how to insert or delete parts of your lesson and how to try out your new lesson. It does cover all aspects of the TUTOR language: that is, what statements to give PLATO but not how to type these statements into a permanent PLATO lesson space. From studying this book you could, in principle, write down on paper a lesson expressed in the TUTOR language, but when you go to a PLATO terminal to type in your new lesson, you may not know what buttons to push to get started. Also, this book discusses TUTOR in more detail than does "Introduction to TUTOR", which makes "The TUTOR Language" less appropriate for your initial study.

It is also hoped that as you study this book you will try things out at a PLATO terminal. TUTOR is designed for interactive use, in which an author writes a short segment of a lesson, tries it, and revises it on the basis of the trial. Normally the sequence write, try, revise, and try again takes only a few minutes at a PLATO terminal. It is far better to create a lesson this way than to write out a complete lesson on paper only to find upon testing that the overall structure is inappropriate.

It is also helpful to try the sample lesson fragments discussed in this book. It is literally impossible to describe fully in this book how the examples would appear on a PLATO terminal. The PLATO medium is far richer than the book medium. One striking example is the PLATO facility for making animations such as a car driving across the screen. As another example, you must experience it directly to appreciate how easy it is at a PLATO terminal to draw a picture on the screen (by moving a cursor and marking points), then let PLATO automatically create the corresponding TUTOR language statements which would produce that picture. PLATO actually writes a lesson segment for you!

This book is written in an informal style. Sometimes, when the context is appropriate, topics are introduced in a different chapter than would be required by strict adherence to a formal classification scheme. In these cases the feature is at least mentioned in the other chapter, and the index at the end of the book provides an extensive cross-linkage. The order of presentation, emphasis, examples, and counter-examples are all based on extensive experience with the kinds of questions working authors tend to ask about TUTOR. In the past no detailed document of this kind was available, which led to various kinds of confusion in the minds of many authors. It is hoped that this book will prevent much confusion from getting started, by answering many questions before they are asked.

If you are a fairly new TUTOR author, read this book lightly to get acquainted with the many features TUTOR offers. Plan to return to the book from time to time as your own authoring activities lead you to seek detailed information and suggestions. Your initial light reading should help orient you to finding appropriate sections for later intensive study. After you feel you know TUTOR inside and out, read this book carefully one last time, looking particularly for links among diverse aspects of the language. This last reading will mean much more to you than the first!

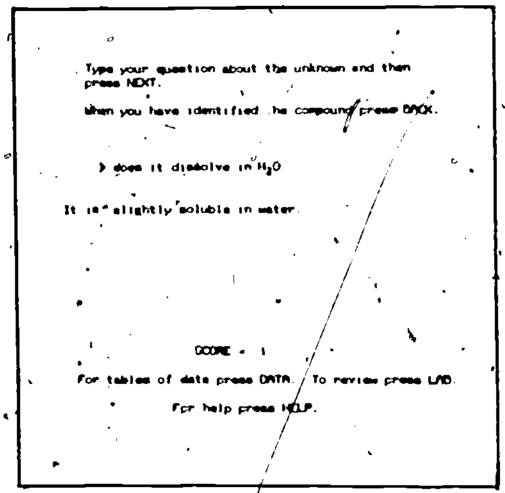
If you are already an experienced TUTOR author, read this book carefully with two goals in mind: to spot features unused in your past work but of potential benefit, and to acquire a more detailed understanding of the structural aspects of the language, with particular emphasis on judging.

The remainder of this introductory chapter contains some interesting examples of existing PLATO lessons, a description of the PLATO keyboard including the use of the special function keys, and a review of the most basic aspects of TUTOR.

Sample PLATO lessons

On the following pages are given several examples of interesting PLATO lessons. All were written in the TUTOR language. They have been chosen to give you some idea of the broad range of possibilities made possible through TUTOR. Each example is illustrated with a photograph of the student's screen at a significant or representative point in the lesson.

The PLATO terminal's display screen consists of a "plasma display panel" which contains 512 horizontal wires and 512 vertical wires mounted on two flat plates of glass between which is neon gas. Any or all of the quarter-million (512×512) intersections of the horizontal and vertical wires can be made to glow as a small orange dot. (The word "plasma" is the scientific name for an ionized gas; the orange glow is emitted by ionized neon gas.) As can be seen in the sample photographs, the PLATO terminal can draw lines and circles on the plasma panel as well as display text using various alphabets. Both drawings and text actually are made up of many dots. TUTOR has many display features for writing or erasing text and drawings on the plasma panel.



Type your question about the unknown and then press NEXT.

When you have identified the compound press BACK.

> does it dissolve in H_2O

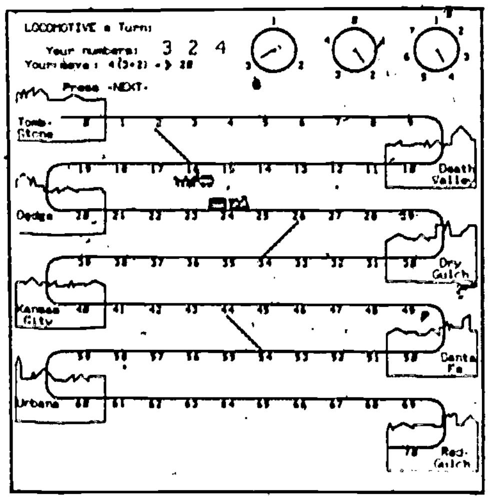
It is slightly soluble in water.

SCORE = 1

For tables of data press DATA. To review press LAB.

For help press HELP.

Dialog in which a chemistry student attempts to identify an unknown compound by asking experimental questions. (Stanley Smith)



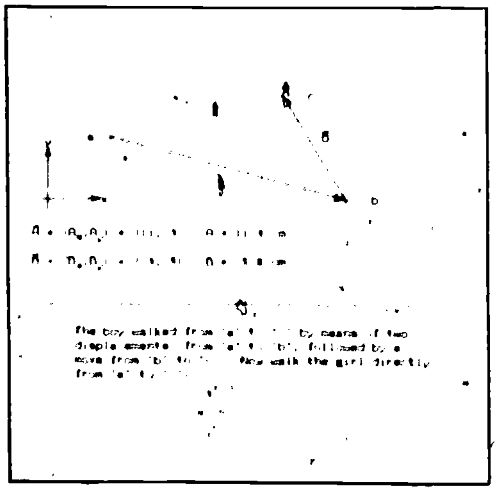
LOCOMOTIVE's Turn:

Your numbers: 3 2 4

Your move: 4(3+2) => 20

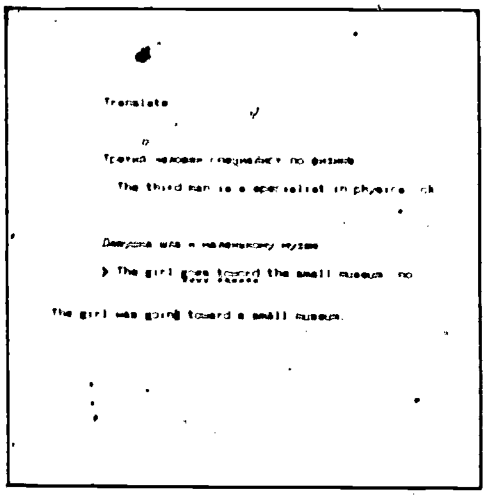
Press -NEXT-

Game of mathematical strategy in which two grade-school children compete in constructing advantageous mathematical expressions from random numbers appearing on the spinners. (Bonnie Anderson)



The boy walked from 'a' to [illegible] by means of two displacements from 'a' to 'b', followed by a move from 'b' to [illegible] Now walk the girl directly from 'a' to [illegible]

Tutorial on vectors in which the student walks a boy and girl around the screen and measures their vector displacements. (Bruce Sherwood)



Translate

Третий человек специалист по физике

The third man is a specialist in physics ok

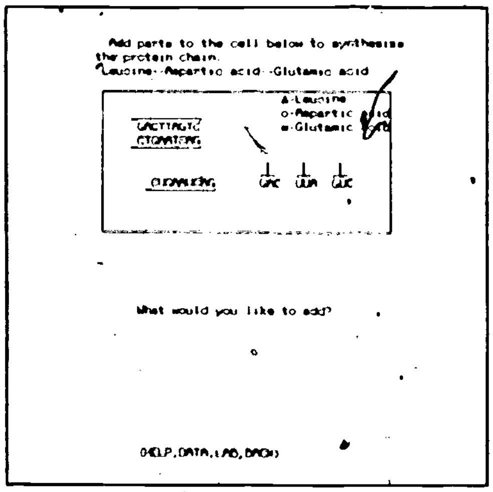
Девушка шла к маленькому музею

> The girl goes toward the small museum no

The girl was going toward a small museum.

Russian sentence drill. The markings under the student's translation of the second sentence indicate incorrect words and misspellings. (Constance Curtin)

These are actual photographs of the plasma panel. The display shows orange text and drawings on a black background, but the pictures are shown here as black on white for ease of reproduction. The plasma panel size is 22 cm. square (8.5 in. square).


Add parts to the cell below to synthesize the protein chain.
Leucine--Aspartic acid--Glutamic acid

△-Leucine
o-Aspartic acid
■-Glutamic acid

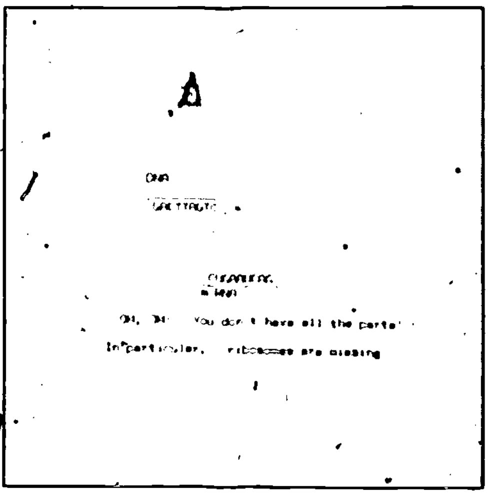
GACTTAGTC
CTGAATCAG

CUGAAUCAG

GAC UUA GUC

What would you like to add?

(HELP, DATA, LAB, BACK)


DNA

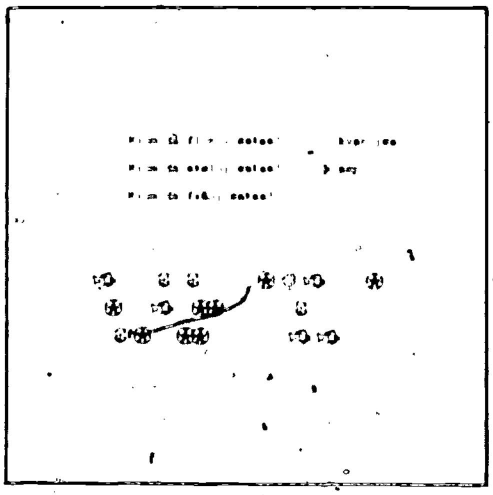
GACTTAGTC

CUGAAUCAG
mRNA

Oh, oh! You don't have all the parts!

In particular, ribosomes are missing

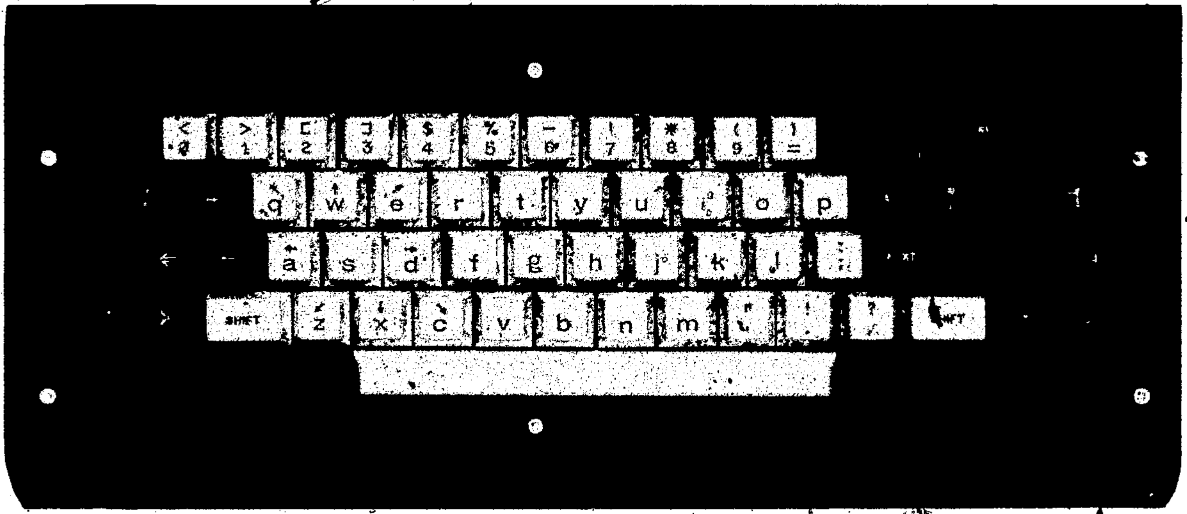
Graphical illustration of the biochemical steps involved in protein synthesis. The student introduces appropriate DNA, RNA, etc., into an initially empty cell, then watches the synthesis proceed. Here the synthesis breaks down for lack of a crucial part. (Paul Tenczar)


[illegible]

[illegible]

[illegible]

Using graphics to teach Esperanto without using English. Here the stars have been circled to emphasize to the student his mistake in counting. (Judith Sherwood)



The PLATO keyboard

Every PLATO terminal has a keyboard like the one pictured above. The keyboard has a number of special features which are closely related to certain aspects of the TUTOR language, such as the HELP key which lets students access optional sections of a lesson, written in TUTOR.

The central white keys include letters, the numbers Ø through 9 along the top row, and punctuation marks. Note that the numbers Ø and 1 are different from the letters o and l. The zero has a slash through it to distinguish it unmistakeably from the letter o. Except for these distinctions, the white keys are the same as the keys on a standard typewriter. Capital letters are typed by pressing either of the SHIFT keys while striking a letter key. Some keys show two different characters, such as the keys in the upper row: depressing a SHIFT key while striking a "4" produces a "$".

Eight of the letter keys (d, e, w, q, a, z, x, and c all clustered around the s key) have arrows marked on them pointing in the eight compass directions. Typing "e" with a SHIFT key depressed normally produces a capital "E" on the screen, not a northeast arrow. The directional arrows are shown because these keys are sometimes used to control the motion of a cursor or pointer on the screen. In this context the student presses an un-shifted "e" and the lesson interprets this as a command to move a cursor northeast on the screen, rather than a command to display an "e" on the screen. Such redefinitions of what a key should do in a particular context provide enormous flexibility. Another interesting example is the use of the keyboard to type Russian text in the Cyrillic alphabet.

Spaces (blank characters) are produced by striking the long "space bar" at the bottom of the keyboard. Holding down a shift key while hitting the space bar produces a backspace. An example of its use is in underlining. The underlined word "cat" is produced by typing c, a, t, backspace, backspace, backspace, underline, underline, underline (underline is shift-6, not to be confused with the minus sign or dash). Typing T, backspace, H will superimpose the two letters: Ħ. Backspace is used for superimposing characters, whereas the ERASE key (just to the right of the letter p) is used to correct typing errors.

A few black keys at the left of the keyboard are mainly associated with mathematical operations: plus, minus (also used as a dash), times and divide (÷ is equivalent to the slash /). The ⇐ is used in TUTOR calculations to assign values to variables. The TAB key is most often used by authors writing lessons rather than by students studying lessons. Its function is similar to the tabulate function on standard typewriters: pressing TAB once is equivalent to hitting the space bar as many times as is necessary to reach a preset column on the screen. Shift-TAB, called CR for "carriage return" in analogy with a typewriter, moves typing down one line and to the left margin. Shift-plus produces a Σ (which means summation in mathematical notation) and shift-minus produces a Δ (which means difference in mathematical notation.)

The black keys at the right of the keyboard are called "function" keys because they carry out various functions rather than displaying a character on the screen. By far the most important function key is NEXT. The cardinal rule for studying PLATO lessons is "When in doubt, press NEXT." Pressing NEXT causes the next logical thing to happen, such as proceeding on to a new display, asking for a response to be judged, erasing an entire incorrect response, etc. The second most important function key is ERASE, which is used to correct typing errors. Each press of ERASE erases one character from the screen. Pressing shift-ERASE (abbreviated as ERASE1) erases an entire word rather than a single character. Note the difference from the backspace (shift-space) which does not erase and is used for superimposing characters.

The EDIT key is also used for correcting typing. Suppose you have typed "the quik brown fox" when you notice the missing "c" in "quik". You could press ERASE1 twice to erase "fox" and "brown", use ERASE to get rid of the "k", then retype "ck brown fox". The EDIT key makes such retyping unnecessary. Instead of hitting ERASE1, you press EDIT which makes the entire sentence disappear. Press EDIT again, and the entire first word "the" appears. Press EDIT again and you see "the quik" on the screen. Use ERASE to correct this to "the quick". Now hit EDIT twice to bring in the words "brown" and "fox". The final result is "the quick brown fox". This takes longer to describe here in words, but pressing the EDIT key a few times is much easier and faster than doing all the retyping that would otherwise be necessary. The EDIT1 key (shift-EDIT) brings back the entire remaining portion of a sentence. For example, after inserting the "c" to make "the quick", you could hit EDIT1 once to bring back "brown fox". You should type some sentences at a PLATO terminal and study the effects produced by EDIT and EDIT1.

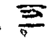
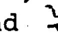
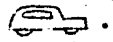
A closely-related key is COPY. COPY and COPY1 are used mainly by authors. While EDIT and EDIT1 cycle through words you have just typed, COPY and COPY1 bring in words from a pre-defined "copy" sentence. These keys are used heavily when changing or inserting portions of a lesson.

The display "a^2b" can be made by hitting a, then SUPER, then 2, then b. SUPER makes a non-locking movement higher on the screen for typing superscripts. Notice that SUPER is struck and released, not held down while typing the superscript. Striking shift-SUPER makes a locking movement, so that the sequence a, shift-SUPER, 2, b will produce a^{2b}. A similar key is SUB: H_2O is made by typing H, SUB, 2, O. A locking subscript results from shift-SUB, which is also what is used to get down from a locking superscript. Similarly, shift-SUPER will move up from a locking subscript.

There are 34 additional characters not shown on the keyboard which are accessible with the MICRO key. For example, striking and releasing the MICRO key followed by hitting "p" produces a π. The sequence MICRO-a produces an α. Typing e, MICRO, q produces è, whereas typing E, MICRO, q produces È. Note the "auto-backspacing" which not only backspaces to superimpose the accent mark but also places the accent mark higher on a capital letter. Six MICRO options involve autobackspacing: ` (q), ´ (e), ¨ (u), ^ (x), ~ (n), and ¸ (c). The last accent mark (MICRO-c) is used for creating cedillas (ç and Ç) and does not involve a different height for capitals. It is easy to remember these keys because of natural associations. The ` and ´ accent marks are on the q and e keys which have the ↖ and ↗ arrows marked on them. The umlaut ¨ usually appear on a u (German ü). The circumflex ^ is on the x key. The tilde ~ usually appears on an n (Spanish ñ). The Greek letters α, β, δ, θ, λ, μ, π, ρ, σ, and ω are produced by typing MICRO followed by a, b, d, t, l, m, p, r, s, or w. Here is a complete list:

key	MICRO-key	key	MICRO-key
		Ø	◁ ("embed" symbols)
		1	▷
a	α (alpha)	< (shift-Ø)	≤ (less than or equal)
b	β (beta)	> (shift-1)	≥ (greater than or equal)
d	δ (delta)	[(shift-2)	{ (braces)
t	θ (theta)	] (shift-3)	}
l	λ (lambda)	$ (shift-4)	# (pound sign)
m	μ (mu)	5	@ (each)
p	π (pi)	6	➤ (arrow)
r	ρ (rho)	=	≠ (not equal)
s	σ (sigma)	) (shift-=)	≡ (identity)
w	ω (omega)	;	~ (approximate)
q	` (grave)	o	° (degree sign)
e	´ (acute)	I	| (vertical line)
c	¸ (cedilla)	D	→ (east)
u	¨ (umlaut)	W	↑ (north)
n	~ (tilde)	A	← (west)
x	^ (circumflex)	X	↓ (south)
		,	⬍ (special)
		+	& (ampersand)
		/	\ (backwards slash)

These are the standard MICRO definitions. You can change these by setting up your own micro table. This is discussed in chapter VIII.

The standard character set includes all the characters we have seen so far, including the Greek letters and other characters accessible through the MICRO key. The shifted MICRO key, called FONT, lets you shift from this standard set of characters to another set of up to 126 special characters which you can design. These special characters might be the Cyrillic, Arabic, or Hebrew alphabet, or they can be pieces of pictures, such as the characters , , and  which form a car when displayed side by side: . Unlike MICRO which only affects the next keypress, FONT locks you in the alternate "font" or character-set. You press FONT again to return to the standard font. The creation of new character sets is described in chapter VIII.

If the author activates it, the ANS key can be used by the student to get the correct answer to a question. This is discussed in chapter VII. The shifted ANS key, TERM, when pressed causes the question "what term?" to appear at the bottom of the screen. At this point you can type any of various keywords to move to a different part of the lesson. The use of TERM is discussed in chapter V.

If you set up an optional help sequence, the student can press the HELP key to enter this sequence. He can press BACK (or BACK1) to go back to where he was when he requested help, or he will be brought back to the original point upon completion of the help sequence. You could also specify a different help sequence accessible by pressing HELP1 (shift-HELP). The six keypresses HELP, HELP1, LAB, LAB1, DATA, and DATA1 can, if activated by the author, allow the student a choice of six different help sequences. You can also activate NEXT, NEXT1, BACK and BACK1, but these simply let the student move around in the lesson without remembering and returning to the original place. In other words, these four keys do not initiate "help" sequences. Usually BACK is reserved for review sequences or similar situations where you want to back up.

The STOP key throws away output destined for the terminal. A useful example is the case of skimming through pages of text in an on-line catalog or collection of notes. If you decide you want to skip immediately to the next page, you might press STOP so as not to wait the several seconds required to finish plotting the present page.

The STOP1 or shift-STOP key plays a crucial role. You press STOP1 to leave a lesson you are studying. When a student is ready to leave the terminal he presses STOP1, which performs a "sign-out" function. Among other things, the sign-out procedure brings his permanent status record up to date so that days later he can sign-in and resume in the same lesson where he left off. When an author presses STOP1 to leave his lesson that he is testing, he is taken back to a point in the PLATO system where he can make changes in his lesson before trying it again.

The key next to HELP with the square (□) on it does not yet have a specific function. The shifted square key is presently used as the ACCESS key, as described in chapter VIII.

Basic aspects of TUTOR

In their simplest form, lessons administered by the PLATO interactive educational system consist of a repeating sequence: a display on the student's screen followed by the student's response to this display. The display information may consist of sentences, line drawings, graphs, animations (moving displays) - nearly anything of a pictorial nature - and in any combination. The student responds to this display by pressing a single key (e.g., the HELP or NEXT key) or by typing a word, sentence, or mathematical expression, or even by making a geometrical construction. Lesson authors provide enough details about the possible student responses so that PLATO can maintain a dialog with student. The sequence of a display followed by a response is the basic building block of a lesson and is called a "unit" in the TUTOR language. This "display-response" terminology is convenient but is not intended to imply that the student is in a subservient position. Often what we will conventionally call the student "response" is a question or a command to PLATO to respond with a display of some kind.

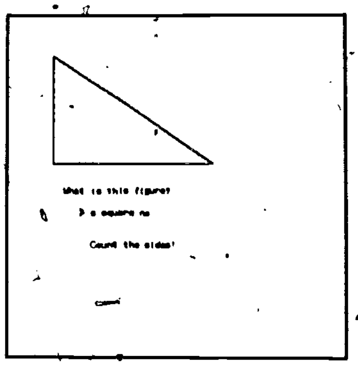
An author constructs a lesson by writing one "unit" at a time. For each unit, the author uses the TUTOR language to specify (1) the display that will appear on the student's screen, (2) how PLATO is to handle student responses to this display, and (3) how the current unit connects to other units.

A statement written in the TUTOR language appears as follows:

$$\underbrace{\texttt{write}}_{\text{command}}\quad\underbrace{\texttt{How are you today?}}_{\text{tag}}$$

The first part of the statement (-write-) is called the <u>command</u>, while the remainder (How are you today?) is called the <u>tag</u>. Command names mnemonically represent PLATO functions. Following is an entire unit written in TUTOR. The figure shows what a student would see on his screen while working on the unit.

```
unit     geometry
at       1812
write    What is this figure?
draw     51Ø;151Ø;154Ø;51Ø
arrow    2Ø15
answer   <it,is,a> (right,rt) triangle
write    Exactly right!
wrong    <it,is,a> square
write    Count the sides!
```


What is this figure?
▸ a square no
Count the sides!
[illegible]

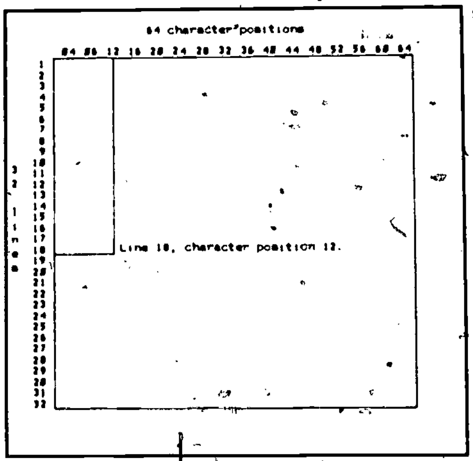
As one can infer, tags individualize commands for the particular function desired. We will discuss each statement of this unit in detail.

```
unit    geometry
```

The -unit- statement initiates each unit. The tag (geometry) will become useful later when units are connected together to form a lesson. Each unit must have a name. No two units in a lesson may have the same name.

```
at      1812
```

The -at- statement specifies at what position on the screen a display will occur. The tag "1812" means that we will display something on the 18th line in the 12th character position. The top line of the screen is line 1 and the bottom line is line 32. There are 64 character positions going from Ø1 at the left edge of the screen to 64 at the right. Thus, 1Ø1 refers to line 1, character position Ø1 (the upper left corner of the screen), while 3264 refers to line 32, character position 64 (the lower right corner of the screen). Note that "Ø" means the number zero, as distinct from the letter "O".


64 character positions
Ø4 Ø8 12 16 20 24 28 32 36 4Ø 44 48 52 56 6Ø 64
1 2 3 4 5 6 7 8 9 1Ø 11 12 13 14 15 16 17 18 19 2Ø 21 22 23 24 25 26 27 28 29 3Ø 31 32
32 lines
Line 18, character position 12.

```
write   What is this figure?
```

The -write- statement causes the text contained in the tag to be displayed on the student's screen. The writing starts at line 18, character position 12, as specified by the preceding -at- statement.

```
draw    51Ø;151Ø;154Ø;51Ø
```

The -draw- statement specifies a straight-line figure to be displayed on the screen. In this particular case a series of straight lines will be drawn starting at location 51Ø (line 5, character position 1Ø), going vertically downward to location 151Ø, then to the right to location 1540, and finally back to the starting point, 51Ø. This produces a right triangle on the student's screen.

```
        arrow   2Ø15
```

The -arrow- statement acts as a boundary-line that separates preceding display statements from following response-handling statements. Thus, what precedes the -arrow- command produces the screen display which remains visible while the student works on the question. Statements after the -arrow- command are used in handling student responses to the display. In addition, the -arrow- statement notifies TUTOR that a student response is required at this point in the lesson. The tag of the -arrow- statement locates the student response on the screen. An arrowhead is shown on the screen at this place to indicate to the student that a response is desired and to tell him where the response will appear. In this case the arrowhead will appear on line 2Ø, character position 15. The student's typing will start at 2Ø17, leaving a space between the arrowhead and his first letter.

```
        answer   <it,is,a> (right,rt) triangle
        .
        .
        .
        .
        wrong    <it,is,a> square
```

The -answer- and -wrong- statements are used to evaluate the student's response. The special brackets < and > enclose optional words, while the parentheses enclose important words which are to be considered synonyms. Thus any of the following student responses would match the -answer- statement: "a right triangle", "it is a rt triangle", "rt triangle", etc.

If the response matches the tag of the -answer- statement, TUTOR writes "ok" after the student's response. For a match to a -wrong- statement, "no" is written. An "ok" judgment allows the student to proceed to the next unit, whereas a "no" judgment requires the student to erase and try again. Any response not foreseen by -answer- or -wrong- statements is judged "no".

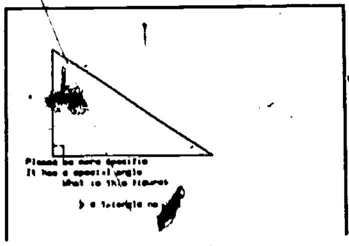
Having matched the student's response, TUTOR proceeds to execute any display statements following the matched -answer- or -wrong- statement. Thus, student responses of "a right triangle" and "square" will trigger appropriate replies. In the absence of specific -at- statements, TUTOR will display these replies three lines below the student's response on the screen. Here is what happens if the student responds with "a lovely tringle, right?":

```
              > a lovely tringle, right?
                  XXXXXX =======<=
```

TUTOR automatically marks up the student's response to give detailed information on what is wrong with the response. The word "lovely" does not belong here and is marked with XXXXXX, the word "tringle" is misspelled and is underlined, and the word "right" is out of order, as is indicated by the small arrow.

Statements can be added to the current example unit which will greatly improve it. Consider the following:

```
unit      geometry
at        1812
write     What is this figure?
draw      51Ø;151Ø;154Ø;51Ø
arrow     2Ø15
specs     bumpshift
answer    <it,is,a> (right,rt) triangle
write     Exactly right!
answer    <it,is,a> (three,3) sided (right,rt) polygon
write     Yes, or a right triangle.
wrong     <it,is,a> triangle
at        16Ø5
write     Please be more specific.
          It has a special angle.
draw      141Ø;1412;1512
wrong     <it,is,a> square
write     Count the sides!
```



Please be more specific
It has a special angle
What is this figure?
▶ a triangle no

As you can see, any number of -answer- and -wrong- statements can be added to the response-handling section of the unit. Time and effort spent by an author in providing for student responses other than the common answer can greatly increase the ability to carry on a personal dialog with each student. The figure shows what the student will see if he responds with "a triangle".

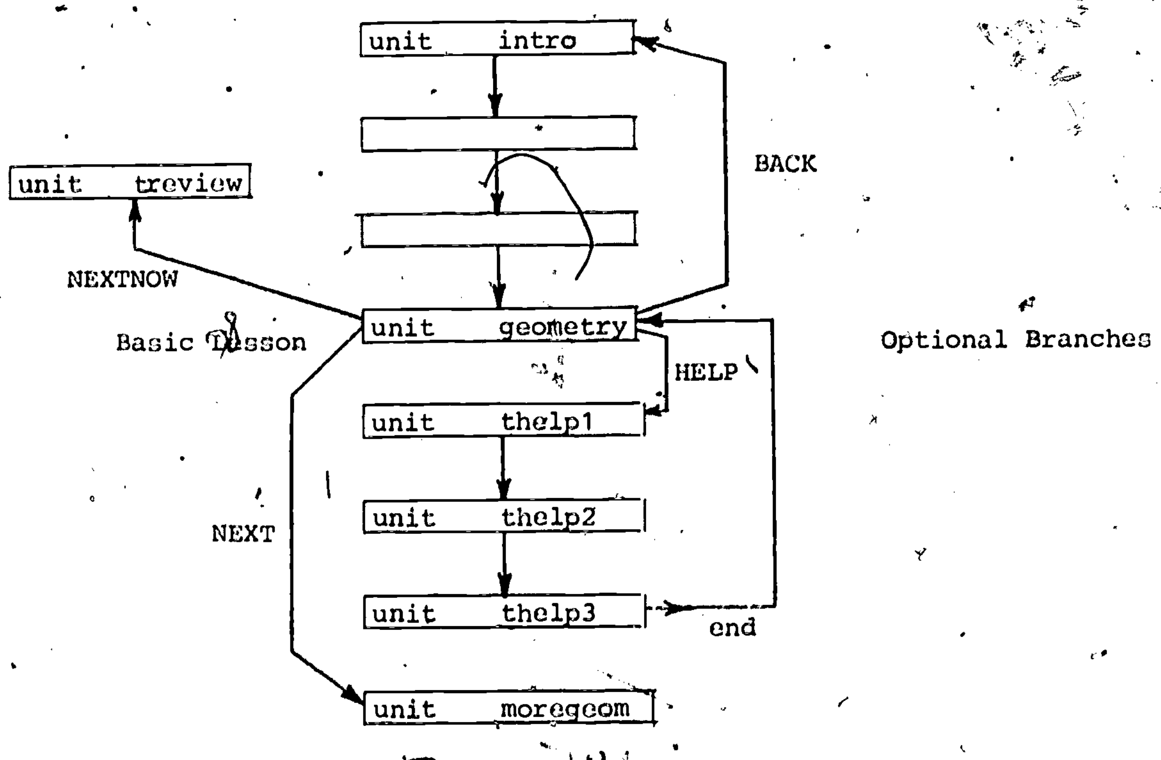
The -specs- statement is introduced here. It is used to give optional specifications on how the student's response is to be handled. In this case the tag, "bumpshift", specifies that any capitalization in the student's response is to be thrown away. As far as the following -answer- and -wrong- commands are concerned, it is as though all the student's capital letters were replaced by lower-case letters. (The student's response displayed on the screen is not changed.) Without this specification, TUTOR would consider "Right Triangle" to be misspelled. There are many convenient options available in a -specs- statement. For example, "specs okextra,noorder" specifies that extra words not mentioned explicitly in following -answer- and -wrong- statements are all right, and that the student's word order need not be the same as the word order of the -answer- and -wrong- statements to achieve a match. Such options can be used to broaden greatly the range of responses which can be handled properly.

Lessons could be written using only the commands already discussed. Explanatory units could be written using only display commands. Tutorial units could be interspersed to test a student's understanding of the lesson material. Thus a single linear chain of units could form a lesson. However, mastery of a few more TUTOR commands opens up a wealth of "branching" or sequencing possibilities. Branching, the technique of allowing alternate paths through a lesson, is one of the keys to personal dialog with each student. The example unit will, therefore, be expanded to include -next-, -nextnow-, -back-, and -help- commands:

```
unit      geometry
next      moregeom
help      thelp1
back      intro
at        1812
write     What is this figure?
draw      51Ø;151Ø;154Ø;51Ø
arrow     2Ø15
specs     bumpshift
answer    <it,is,a> (right,rt) triangle
write     Exactly right!
wrong     <it,is,a> triangle
at        16Ø5
write     Please be more specific.
          It has a special angle.
draw      141Ø;1412;1512
wrong     <it,is,a> square
nextnow   treview
```

The tag of the -next- statement following the -unit- command gives the name of the next unit the student will see upon the successful completion of unit "geometry". The -next- statement is necessary because in a highly-branching lesson sequence the next unit for a student may not be the unit following in the lesson. For example, a diagram of the lesson flow involving unit "geometry" might be:

Partial Diagram of Lesson



In moving from one unit to another the screen normally is automatically erased to make room for the displays produced by the following unit.

The -help- statement refers to a help unit which the student may reach through use of the HELP key. Help units are constructed in the same manner as unit "geometry". However, the last (or only) unit in a help sequence is terminated by an -end- command. Upon completing the last help unit, the student is returned to the "base" unit, the unit from which he branched (in this case unit "geometry"). The student need not complete the entire help sequence. He may press BACK or shift-BACK to return to the base unit from any point in the help sequence. Help units for unit "geometry" could appear as follows:

```
        :
        * These units are help units for "geometry".
        unit    thelp1
        at      1828
        write   The figure has three sides.
        draw    51Ø;151Ø;154Ø;51Ø
        *

        unit    thelp2
        at      1828
        write   It also has three angles.
        draw    51Ø;151Ø;154Ø;51Ø
        *

        unit    thelp3
        at      1828
        write   Note the right angle.
        draw    51Ø;151Ø;154Ø;51Ø
        end
```

Any statement which begins with an asterisk (*) has no effect on the operation of the lesson and may be used anywhere to insert comments to describe the units. A comment statement between units improves readability by guiding the eye to the unit subdivisions of the lesson.

The -back- statement permits the student to move to a different unit by pressing the BACK key. Because of its name, it is customary to associate a review sequence with the BACK key. If a student is in a non-help unit that does not contain a -back- statement, the BACK key does nothing. In a help-sequence unit that has no -back- statement, the BACK key returns the student to the original base unit.

If the student calls the figure "a square", he will see this response judged "no" and gets the reply "Count the sides!" The -nextnow- statement is used to force the student through additional material. It locks the keyboard so that only the NEXT key has any effect. In particular, the student cannot erase his response. When he presses NEXT, he will be sent to unit "treview". Upon completion of one or more units of review about triangles, the author might return the student to unit "geometry". Thus, this student's lesson flow might consist of:

1) a discussion of geometric figures;

2) a question about a right triangle;

3) an error causing -nextnow- to lock the keyboard;

4) further study of triangles;

5) finally, a return to the right triangle.

Consider now the problem of using unit "geometry" for a second student response. Additional display information is needed to ask the student a second question and another -arrow- command is needed plus a second set of response-handling statements. The unit could appear as follows:

```
unit     geometry
next     moregeom
back     intro
help     thelp1
at       1812
write    What is this figure?
draw     51Ø;151Ø;154Ø;51Ø
arrow    2Ø15
  .    } Response-handling statements
  .    } for first arrow.
  .    }
  .
endarrow
at       2512
write    How many degrees in
         a right angle?
help     angles
arrow    2815
  .    } Response-handling statements
  .    } for second arrow
  .    }
```

The -endarrow- command delimits the response-handling statements associated with the first -arrow-. Only when the first -arrow- is satisfied by an "ok" judgment will TUTOR proceed past the -endarrow- command to present the second question. The statement "help angles" overrides the earlier statement "help thelp1". If the student presses the HELP key while working on the second -arrow- he will reach unit "angles" rather than unit "thelp1".

The second question could have been given in a separate unit rather than following an -endarrow- command. The major difference is that the entire screen is normally erased in proceeding to a new unit, whereas here the second question was merely added to the existing screen display. Even if there is only one -arrow- command in a unit, -endarrow- can be useful, for it can be followed by display or other statements to be performed only after the -arrow- is satisfied. This is particularly convenient if there are several -answer- commands corresponding to several different classes of acceptable responses.

Fourteen TUTOR commands have been illustrated in this chapter. This repertoire is adequate to begin lesson writing; and if you have access to a PLATO terminal, it would be useful at this point to try out the ideas discussed so far.

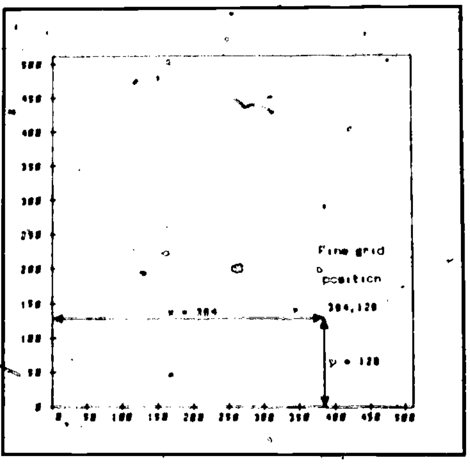
II. More on Creating Displays

Particular attention should be paid to the question of how to display text and line drawings to the student. Good or poor displays of material in a lesson can make the difference between a successful or unsuccessful lesson. Imaginative use of graphics, including animations (moving displays), will capture the attention of the student and transmit your message to him much more efficiently than would mere text. You have already seen how to write text and draw figures by using the -at-, -write-, and -draw- commands. This chapter will discuss how to achieve finer control over screen positions, how to draw circles and circular arcs, how to display large-size text and write at an angle, and how to erase portions of the screen. The ability to erase a portion of the screen makes it possible to create animated displays.

Coarse grid and fine grid

It is convenient to specify a line number and character position for displaying text. We have seen that the TUTOR statement "at 1812" instructs PLATO to display information starting on the 18th line at the 12 character position. Line 1 is at the top of the screen and line 32 is at the bottom. Each line has room for 64 characters, with character position Ø1 at the left and character position 64 at the right. This numbering scheme is called the coarse grid or gross grid.

Sometimes it is necessary to position text or draw a figure with finer control than is permitted by the coarse grid. The PLATO screen consists of a grid of 512 by 512 dots, and the position of any of these quarter-million dots can be specified by giving two numbers - the number of dots from the left edge of the screen (often called "x") and the number of dots up from the bottom of the screen (often called "y"):



The position shown would be referred to as "384,128" in an -at- or -draw- statement. This position is equivalent to the coarse grid location 2449 (line 24, character position 49). As an example,

```
unit    double
at      384,128
write   DOUBLE WRITING
at      385,129
write   DOUBLE WRITING
```

would write "DOUBLE WRITING" twice, displaced horizontally and vertically by one dot. This looks like this:

(Greatly enlarged.)

The -draw- command permits mixing the two numbering schemes

```
draw    1215;1225;12Ø,24Ø;1855
```

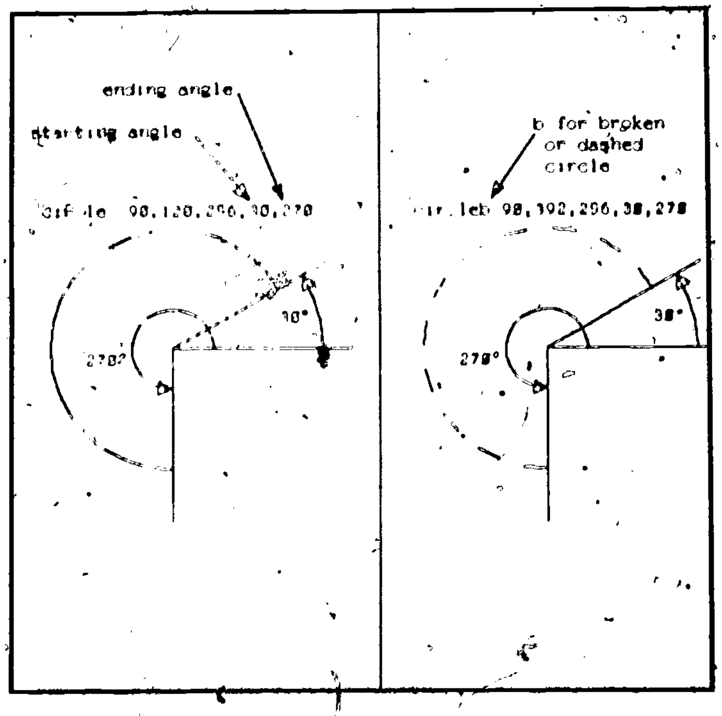
This means "draw a straight line from 1215 to 1225, draw a second straight line from there to (12Ø,24Ø), then draw a third straight line from there to 1855". Note that each point, whether expressed in coarse grid or fine grid, must be set off by a semicolon.

The -circle- command

In addition to displaying text and line drawings, it is possible to draw circles, parts of circles, and broken or dashed circles. A circle whose radius is 125 dots, centered at x=2ØØ, y=3ØØ, is specified by

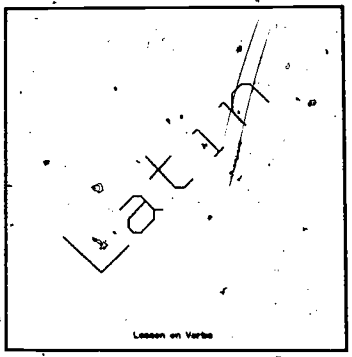
$$\text{circle}\quad \underbrace{125}_{\text{radius}},\underbrace{2ØØ,3ØØ}_{\text{center (x and y)}}$$

If the command name is changed to -circleb- a broken or dashed circle will be drawn. A partial circle is specified by giving a starting and ending angle:



Large-size writing: -size- and -rotate-

It is possible to have text displayed in larger than normal size, and even write at an angle. This is particularly useful in showing an eye-catching title on a page. Here is a sample display with the corresponding TUTOR statements. The "$$" permits a comment to appear after a tag.




```
unit      title
size      9.5       $$ text 9.5 times normal size
rotate    45        $$ text rotated 45 degrees
at        2519
write     Latin
size      Ø         $$ return to normal writing
rotate    Ø
at        3125
write     Lesson on Verbs
```

For technical reasons the large-size writing comes on the screen much more slowly than does normal text, but the speed is adequate for short titles. Use "size Ø" to return to normal writing. Normal writing is unaffected by -rotate-: use "size 1" if it is desired to rotate text of the standard size. Size 1 writing appears at the same slow speed as larger writing (about 6 character per second, or 6Ø words per minute). Only size Ø writing is rapid (18Ø characters per second, or 18ØØ words per minute).

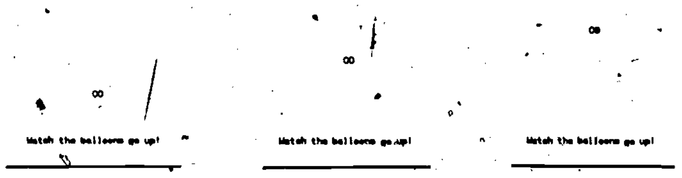
BE SURE TO RETURN TO SIZE Ø!! If you forget to place a "size Ø" statement after the completion of the special writing, all of your text will be written slowly (and possibly rotated). It is also good practice to say "rotate Ø", so that the next time you use "size" the rotation will be through Ø degrees unless stated otherwise.

Animations (moving displays): -erase- and -pause-

An animated display can be created by repetitively displaying some text, pausing, erasing the text and rewriting it in a new position on the screen.

Here is a unit which will show two balloons floating upwards:

```
        unit     balloons
        at       3Ø2Ø
        write    Watch the balloons go up!
        at       25Ø,1ØØ
        write    00          $$ use 00 for balloons
        pause    1.5         $$ suspend processing for 1.5 seconds
        at       25Ø,1ØØ
        erase    2           $$ erase two characters
        at       25Ø,15Ø     $$ reposition 5Ø dots higher
        write    00
        pause    1.5
        at       25Ø,15Ø
        erase    2
        at       25Ø,2ØØ
        write    00
        pause    1.5
          .
          .
          .
```


00
Watch the balloons go up!
00
Watch the balloons go up!
00
Watch the balloons go up!

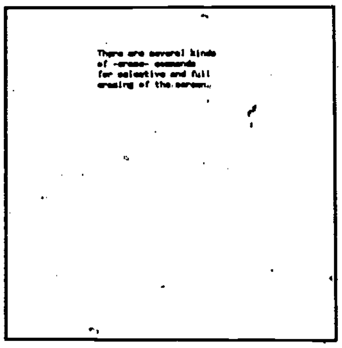
The statement "erase 2" selectively erases two character positions without disturbing the rest of the screen. In particular, the text "Watch the balloons go up!" will stay on the screen.

There are other forms of the -erase- command. The statement "erase 12,3" will selectively erase a block of 12 character positions on three consecutive coarse-grid lines. The statement "erase" with no tag will erase the entire screen instantaneously: the same full-screen erase normally takes place automatically upon moving to a new main unit.

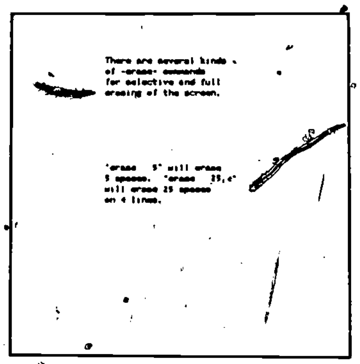
-pause-, -time-, and -catchup-

The -pause- statement with a tag in seconds suspends processing for the specified amount of time. If the tag is omitted, TUTOR waits for the student to strike a key, any key, rather than wait a specified amount of time. This form is particularly suitable in more complicated situations where the student may want to study each step before proceeding. Here is an example:

```
unit    discuss
at      52Ø
write   There are several kinds
        of -erase- commands
        for seleotive and full
        erasing of the screen.
pause
at      152Ø
write   "erase   5" will erase
        5 spaces.  "erase   25,4"
        will erase 25 spaces
        on 4 lines.
pause
at      252Ø
write   An -erase- command
        with a blank tag
        will erase the whole
        screen.
```

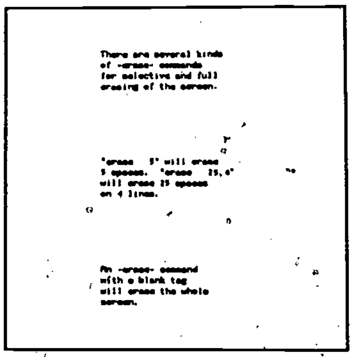


```
There are several kinds
of -erase- commands
for selective and full
erasing of the screen.
```



```
There are several kinds
of -erase- commands
for selective and full
erasing of the screen.

"erase   5" will erase
5 spaces.  "erase   25,4"
will erase 25 spaces
on 4 lines.
```



```
There are several kinds
of -erase- commands
for selective and full
erasing of the screen.

"erase   5" will erase
5 spaces.  "erase   25,4"
will erase 25 spaces
on 4 lines.

An -erase- command
with a blank tag
will erase the whole
screen.
```

Each time the student presses a key to move past the -pause- command, more text is added to the screen. This prevents the student from feeling overwhelmed by too much text being thrown at him all at once. Each new paragraph is added only when he signals by pressing a key that he wants to go on. On the other hand, this structure leaves the earlier paragraphs on the screen so that he can look back to review. If the -pause- commands were replaced by -unit- commands, each paragraph would reside in a separate main unit. When the student presses NEXT to move on to the next main unit, the screen is completely erased to make room for the next display. This would accomplish the objective of letting the student control the rate of presentation of new material but would not leave the earlier paragraphs on the screen for review and comparison.

It is inadvisable in this application to use "pause 15" rather than "pause", for then the student has no control over the presentation rate. Any time delay you choose will be too fast for some students and too slow for others. A timed -pause- is mainly useful for animations. Sometimes it is appropriate to move on after a long time if the student hasn't pressed a key himself. This can be achieved with a -time- command:

```
    .
    .
    .
    time    3Ø.
    pause
    .
    .
    .
```

The "time 3Ø" statement will "press the timeup key" after 3Ø seconds, so that if the student does not press a key, TUTOR will. But the student can move on sooner by pressing a key before then, which is not possible if you use "pause 3Ø".

To summarize, there are three types of -pause- situations:

```
    1)  pause   n     pause n seconds whether
                      keys are pressed or not

    2)  pause         wait for any key

    3)  time    n     wait for any key or n seconds
        pause
```

Occasionally you might want to send several seconds worth of output to the student's screen, then pause two seconds, then add something else. If you write

```
    several seconds of display---text and
    drawings which take several seconds to
    paint on the screen

              -followed by-

    pause   2
    write   More text. . . .
```

you will not get the desired effect because TUTOR will add "More text..." right after the initial material headed toward the terminal, since the "pause 2" ends before the initial display is finished. The student will see no gap between the first and second parts of the display. The problem is solved with a -catchup- command:

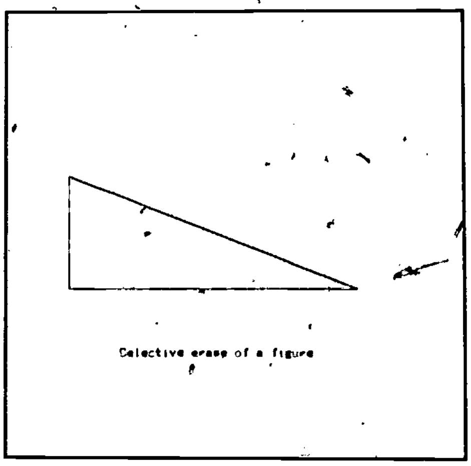
```
          .
          .
          .
          catchup
          pause    2
          write    More text. . . .
```

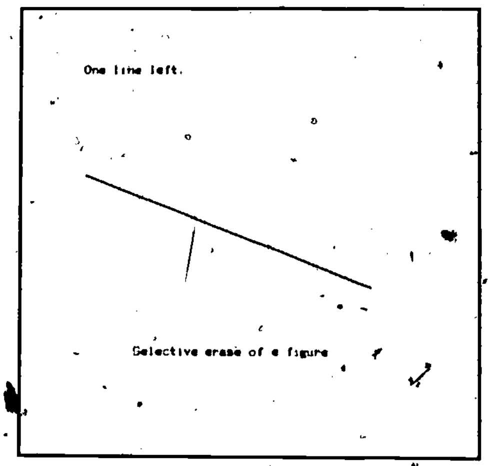
The -catchup- command tells TUTOR to let the terminal "catch up" on its work up to that point before continuing. Then you pause an additional two seconds, and you get the desired effect.

The -mode- command

The -erase- command may be used to erase blocks of character positions or the whole screen. Something else is needed for selectively erasing line drawings created with -draw- and -circle- statements. The PLATO terminal can be placed in an erasure mode in which the terminal interprets all display instructions as requests to erase rather than to light up the corresponding screen dots. This is done with the -mode- command:

```
          unit     modes
          at       2517
          write    Selective erase of a figure
          draw     1210;2010;2050;1210        $$ triangle
          pause                               $$ wait for a key
          mode     erase
          draw     1210;2010;2050             $$ part of the triangle
          mode     write
          at       510
          write    One line left.
```





The "write" mode is the normal display mode. Be sure to specify "mode write" when you are through with "mode erase", or all further writing in that unit will be invisible!

In the standard mode ("write") it is possible to superimpose or overstrike text with another -write- statement. If, however, a "mode rewrite" statement is executed the second -write- statement will erase the previous text as it writes the new text, and there will be no superposition. Compare these sequences in write and rewrite modes:

```
mode    write                    mode    rewrite
at      1215                     at      1215
write   ABC                      write   ABC
at      1215                     at      1215
write   abc                      write   abc
  |                                |
  |       write mode               |        rewrite mode
  v                                v
 ABC                              abc
(superimposed)                  (not superimposed)
```

In the rewrite case the second -write- statement wipes out the 3-character area as it writes the new information. Each character area is 8 dots wide by 16 dots high. This determines the number of rows and columns in coarse grid: (512/8)=64 characters fit across the screen, and (512/16)=32 lines of characters fill the screen vertically.

The statement "erase 2" is actually equivalent to

```
mode    rewrite
write   (two spaces)
mode    (previous mode)
```

Writing spaces (blank characters) in rewrite mode wipes out an entire character area.

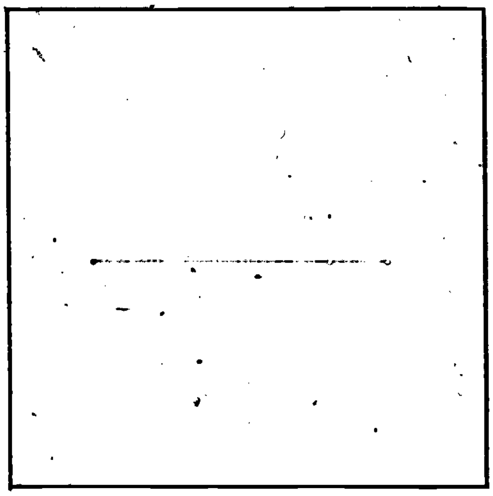
The balloon animation could have been written

```
.
.
.
at      25ø,1øø
write   00
pause   1.5
mode    erase
at      25ø,1øø
write   00          $$ instead of "erase 2"
mode    write
.
.
.
```

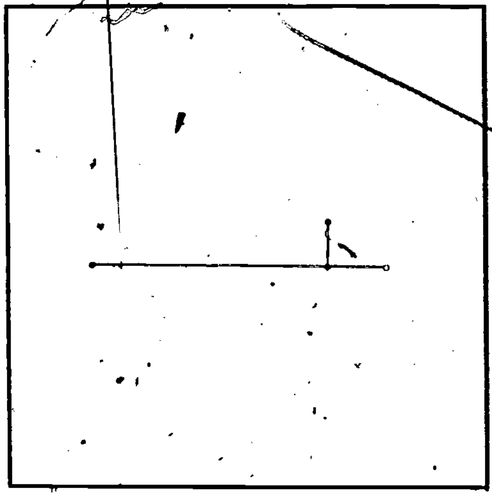
This form would be different from the form using "erase 2" if there were other screen dots lit in this area. "erase 2" completely erases two character positions while "write 00" in the erase mode erases only the dots that make up the letters "00" without disturbing neighboring dots.

Automated display generation

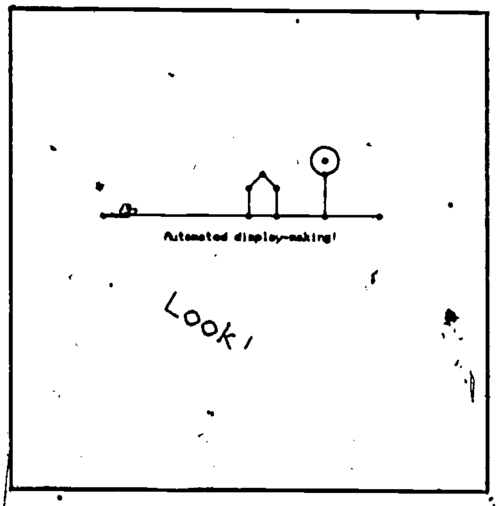
It should be mentioned that an author working at a PLATO terminal can use a moving cursor to design a display involving text, line figures, circles and arcs. The PLATO system then automatically creates corresponding TUTOR statements which would produce that display. The author can alter these statements, convert them back into a display, and add to or alter the resulting display. This facility makes it unnecessary in most cases to worry about the details of screen positions. Here is an example of such operations:



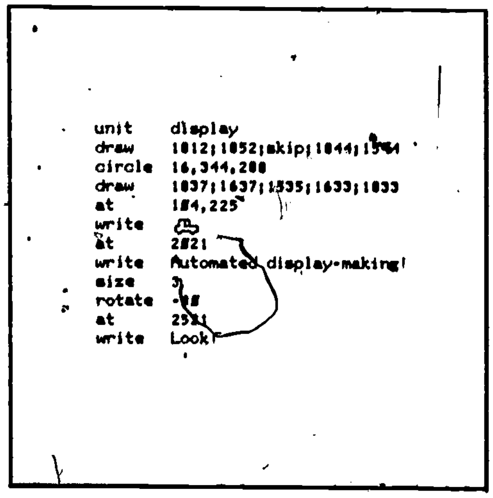
Move the cursor (the "+") to draw the road and to mark the ends of the tree trunk.



Draw the tree trunk.

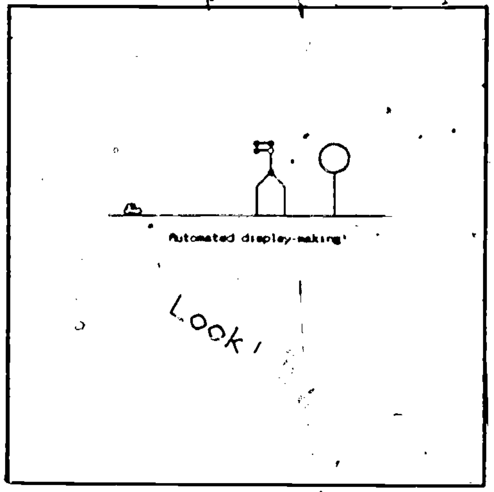


Specify a circle for the top of the tree. Draw the house. Place text of various kinds on the screen. (The car uses special characters.)

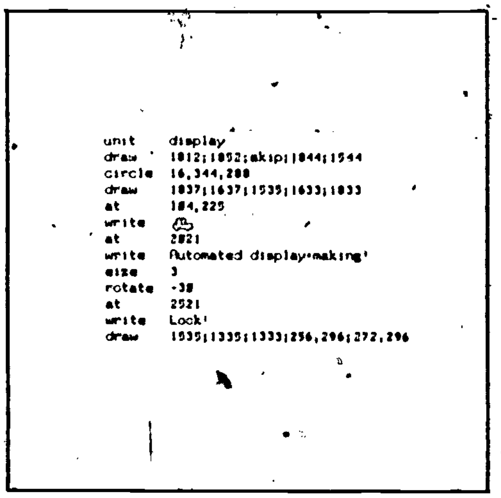


```
unit      display
draw      1012;1052;skip;1044;1544
circle    16,344,200
draw      1037;1637;1535;1633;1033
at        104,225
write     
at        2021
write     Automated display-making!
size      3
rotate    -[illegible]
at        2521
write     Look!
```

PLATO automatically generates TUTOR statements corresponding to the desired display!

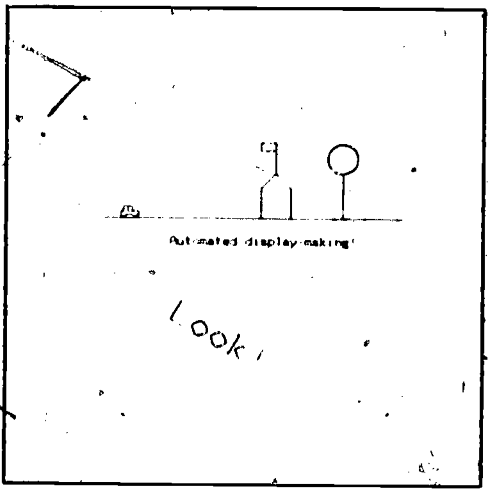


Recall the display and add a flag to the house.



```
unit    display
draw    1012;1092;skip;1044;1544
circle  16,344,200
draw    1037;1637;1535;1633;1033
at      104,225
write   
at      2021
write   Automated display-making'
size    3
rotate  -30
at      2521
write   Look!
draw    1535;1335;1333;256,296;272,296
```

PLATO appends a -draw- statement corresponding to the flag.



Final result. The illustrations in this book were created by these techniques. The screen displays were photographed.

III. Building your own tools: the -do- command

You now know enough about presenting material to the student to be able to make handsome displays. You will be able to do even more when you learn how to tell PLATO to calculate complicated displays for you. Before discussing how to do calculations we will pause to introduce an extremely important concept, the "subroutine", which is fundamental to all aspects of authoring. We will apply the concept of a subroutine right away to certain display problems.

To introduce the use of subroutines, consider the problem of placing some standard message on several of your main lesson pages. For example, in the many units where you make help available to the student (if he presses the HELP key) you might like to advertise this fact by placing at the bottom of the page this display:

HELP is available

The corresponding TUTOR statements might be

```
        at      3123
        write   HELP is available
        draw    3022;3041;3141;3122;3022
```

It would be tedious to copy these statements into every unit where they were required. Moreover, if you decided later to move this to the upper right corner of the screen, you would have to find all occurrences of this and change all of them. There is a way around these difficulties, and in later work we will find further important advantages to the method. Suppose we write a "subroutine", a unit to be used many times as needed:

```
        unit    helper
        at      3123
        write   HELP is available
        draw    3022;3041;3141;3122;3022
```

Where we need to show this message we need only write the statement

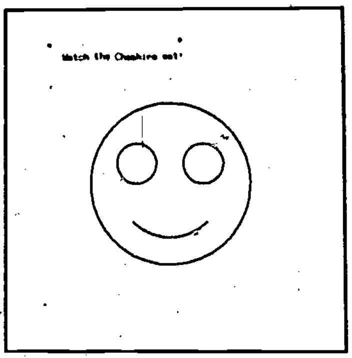
```
        do      helper
```

which attaches unit "helper" to the present unit. It is as though we had inserted the contents of unit "helper" at the point where we say "do helper". Now, instead of a dozen copies of the display statements we have only one, plus a dozen -do- commands. The -do- command may appear anywhere in a unit: where you put it will determine when the associated display appears on the screen in relation to your existing display material. All these displays may be changed by simply changing the subroutine unit! It is not necessary to change the -do- statements; just change unit "helper" which they all use.

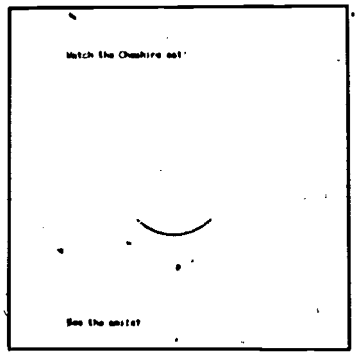
The use of -do- improves the readability of a TUTOR lesson. When you see "do helper" anywhere in your lesson you recognize at a glance what it is for, whereas the contents of unit "helper" might contain a large number of statements which would clutter up your other units and decrease readability if these statements appeared directly in each unit.

Let's consider another use. Suppose we wish to draw a "Cheshire cat" which fades to a smile as Alice watches. We want to draw a cat face made up of the smile plus all the rest of the face, then erase everything but the smile. Here is an elegant way to do it:

```
unit     Alice
at       512
write    Watch the Cheshire cat!
do       cat
catchup             $$ wait for cat to be drawn
pause    4          $$ then pause 4 seconds
mode     erase
do       face
mode     write
at       3Ø12
write    See the smile?
```



We will need some units to use as the subroutines:



```
unit     cat
do       face
do       smile
*
unit     face
circle   12Ø,25Ø,25Ø        $$ outline
circle   3Ø,2ØØ,28Ø         $$ eye
circle   3Ø,3ØØ,28Ø         $$ eye
*
unit     smile
circle   8Ø,25Ø,25Ø,225,315     $$ smile - arc goes from
                                       225° to 315°
```

Note that unit "Alice" does unit "cat", which in turn does units "face" and "smile". TUTOR permits you to go ten levels deep in -do-s. Here we have gone only two levels deep. Note that unit "smile" on its own is a useful subroutine and might be done whenever just the smile is desired.

To summarize, we can build useful tools by constructing "subroutines"--units which may be done from many places in the lesson. The liberal use of -do- improves readability, reduces typing, and facilitates revising the lesson. This last point is particularly important when there is a "bug" (unknown error) in the lesson. Debugging becomes vastly simpler because of the modular nature of subroutines and the localization of critical points in a lesson which uses -do- extensively.

IV. Doing calculations in TUTOR

You can make TUTOR calculate things for you. For example:

```
        .
        .
        .
at      12∅1
write   Who is buried
        in Grant's tomb?
arrow   12∅1+3∅8
        .
        .
        .
```

The -arrow- statement as written is completely equivalent to "arrow   15∅9". Or consider this:

```
circle  (41²+72.6²)^1/2,1∅∅,2∅∅
```

$$\text{circle}\quad (41^2+72.6^2)^{1/2},1\emptyset\emptyset,2\emptyset\emptyset$$

The radius of the circle will be taken to be the square root of the sum of 41 squared and 72.6 squared.

Just about any expression that would have made sense to your high school algebra teacher will be understood and correctly evaluated. Some additional examples:

Expression	TUTOR Evaluation	
$3.4+5(2^3-3)/2$	15.9	
$2\times3+8$	14	(NOT 22)
$\sin(3\emptyset^\circ)$	∅.5	(See Appendix C for other functions.)
$49^{1/2}$	7	
$(4+7)(3+6)$	99	
$6/5\times1\emptyset^{-3}$	.12∅∅	(NOT $1.2\times1\emptyset^{-3}$)

If your high school algebra is rusty, we remind you that "2×5+3" means "(2×5)+3" which is 13, not "2×(5+3)" which is 16. The rule is that multiplication is "more important" than addition or subtraction and gets done first. If at some point you are unsure, just use plenty of parentheses around portions of your expression to make the meaning unambiguous.

A similar point holds for division, which is considered "more important" than addition or subtraction. "8+6/2" means "8+(6/2)" which is 11, not "(8+6)/2" which would be 7. The only ticklish point is whether multiplication is more or less "important" than division. TUTOR agrees with most mathematical books and journals that multiplication is more important than division, so that "6×4/3×2" means "(6×4)/(3×2)" which is 4. Note that this means that TUTOR considers "1/2(6+4)" to be "1/(2(6+4))" which is ∅.∅5, not "(1/2)(6+4)" which would be 5. Again, when in doubt use parentheses. You could write ".5(6+4)" if you wish, which is unambiguous.

Experience has shown that students tend to write algebraic responses according to these rules, and making TUTOR conform to these rules facilitates the correct judging of student algebraic and numerical responses.

Having seen how expressions are handled, we can introduce "student variables" which may be used to hold numerical values obtained by evaluating expressions. These stored results can be used later in the lesson. For example, a "variable" might hold the student's score on a diagnostic quiz, and this score could be used later to determine how much drill to give the student. The storage place is called a "variable" because what it holds may vary at different times in the lesson. Another variable might count the number of times the student has requested help, in which case the number which it holds would vary from ∅ to 1 to 2, etc.

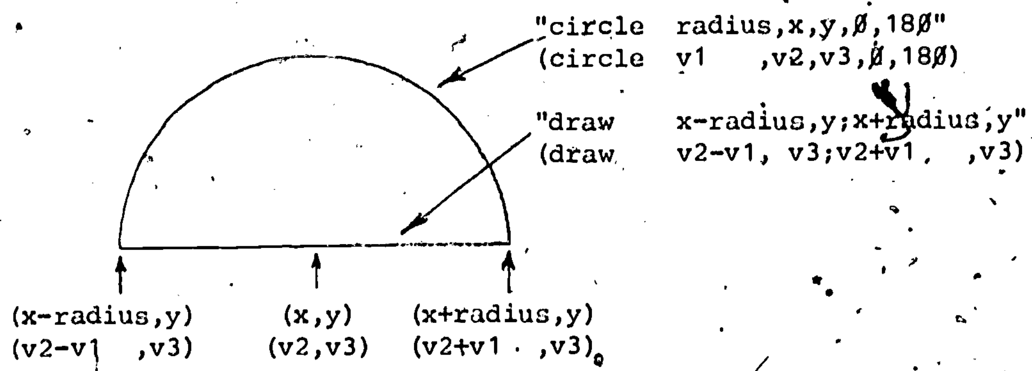
There are 15∅ "student variables" which can be used for storing up to 15∅ numerical values. These "student variables" are unimaginatively called

v1, v2, v3,...v148, v149, v15∅.

Later in this section we will learn how to give them names which are appropriate to their particular uses in a particular lesson, such as "radius", "wrongs", "tries", "speed", etc. But at first we'll just use their primitive names, v1 through v15∅.

These variables are called student variables because each of the many students who may simultaneously be studying your lesson has his or her own private set of 15∅ variables. You might use variable v23 to count the number of correct responses on a certain topic, which will be different for each student. If there are forty students working on your lesson, TUTOR is keeping track of forty different "v23's", each one different. This is done automatically for you, so that you can write the lesson with one individual student in mind, and v23 may be considered simply as containing that individual student's count of correct responses. Thus one student might be sent to a remedial unit because the contents of his variable number 23 show that he did poorly on this topic. Another student might be jumped ahead because the contents of her variable 23 indicate an excellent grasp of the material. It is through manipulation of the student variables that a lesson can be highly individualized for each student.

Variables are useful in building certain kinds of displays. Let's see how to build a subroutine which can draw a half-circle in various sizes, depending on variables which we set up.



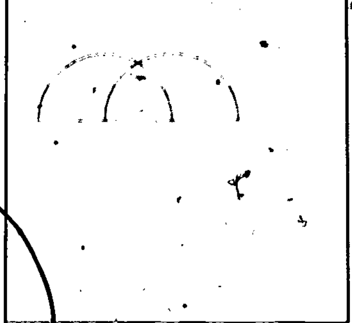
In order to specify the size of the figure and its location on the screen, we must specify a radius and a center (x and y). If we let variable v1 hold the value for the radius, and let v2 and v3 keep the horizontal x and vertical y positions of the center, we can draw such a figure with the following unit:

```
unit     halfcirc
circle   v1,v2,v3,∅,18∅          $$ 18∅ degree arc
draw     v2-v1,v3;v2+v1,v3       $$ horizontal line
```

In order to use this subroutine we might write

```
unit    vary
calc    v1⇐1∅∅                   $$ radius 1∅∅
calc    v2⇐15∅                   $$ x center at 15∅
calc    v3⇐3∅∅                   $$ y center at 3∅∅
do      halfcirc
calc    v2⇐v2+v1                 $$ increment x center
do      halfcirc                 $$ radius and y unchanged
```

The statement "calc v2⇐15∅" means "perform a calculation to put the number 15∅ in variable v2". The statement "calc v2⇐v2+v1" means "calculate the sum of the numbers presently held in variables v2 and v1, and put the result in variable v2". In the present case this operation will store the number 25∅ (15∅+1∅∅) in variable v2 for use in the second "do halfcirc". Note that the second "do halfcirc" will use the original values of v1 and v3, which have not been changed. This unit will produce this picture:



The ⇐ symbol is called the "assignment" symbol, because it assigns a numerical value to the variable on its left. This numerical value is obtained by evaluating the expression to the right of the assignment symbol.

A slightly more complicated example of a -calc- statement is

```
        calc    v3⇐5v2+v1
```

which means "multiply by 5 the number currently held in v2, add this to the number held in v1, and store the result in v3." In conversation you might read this as "calc v3 assigned five v2 plus v1" or "calc v3 becomes five v2 plus v1". Notice that it is common practice to refer simply to "v2" when we really mean "the number currently held in variable v2".

The simplest possible -calc- statement merely assigns a number to a variable, as in "calc v2⇐15∅". It is permissible to make more than one assignment in a -calc-:

```
        calc    v3⇐v7⇐18.6²
```

This will assign the value 18.6^2 to both variables v3 and v7.

<u>Giving names to variables: -define-</u>

Your programming can be made much more readable by "defining" suitable names for the student variables which you use. For example, in the units just discussed, the quantities of interest were the radius and center (x and y) of the circular arc. We should <u>precede</u> such units with a -define- statement:

```
        define  radius=v1              $$ names may be 7 characters long
                x=v2,y=v3
        unit    vary
        calc    radius⇐1∅∅
                x⇐15∅                  $$ The command name -calc-
                y⇐3∅∅                  $$ may be omitted on successive lines
        do      halfcirc
        calc    x⇐x+radius
        do      halfcirc
        *
        unit    halfcirc
        circle  radius,x,y,∅,18∅
        draw    x-radius,y;x+radius,y
```

The -define- statement tells TUTOR how to interpret the defined names when they are encountered later in expressions. The units are now much more readable than they were when we used v1, v2 and v3.

Giving meaningful names to the variables you use is very important! After an absence of several months you yourself would have difficulty in remembering what you are keeping in, say, variable v26, whereas the name "tries" would remind you immediately that this holds a count of the number of times the student has tried to answer the question. The importance of readability is even more vital if a colleague is working with you on the material. He would find it extremely frustrating to try to figure out what you are keeping in v26. So,

USE -DEFINE-!

There should not be any v3's or v26's anywhere in your lesson except in the -define- statement itself. Put all your definitions at the very beginning of the lesson where you will have ready reference to which variables you are using.

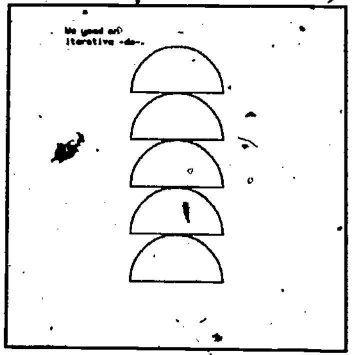
The only reason we started out using the primitive v-names was to give a more concrete feeling for the meaning of a student variable. From here on we will use defined variable names. A preceding -define- statement is assumed.

> WARNING: Normal algebraic notation permits expressions such as "rcosθ", but in TUTOR you must write "r×cos(θ)" or "r(cos(θ))". That is, you must use an explicit multiplication sign between names (either your defined names such as "r" or TUTOR-defined names such as "cos"), and you must place parentheses around the arguments of functions - the "θ" in cos(θ).

The reason for this is that TUTOR cannot cope with the ambiguities of trying to decide whether an expression such as "abc" means "a×bc" (if there is a name "bc"), or "ab×c" (if there is a name "ab"), etc. Later, when we discuss the important topic of judging student responses, we will see that TUTOR can make reasonable guesses when treating a student's algebraic response and can permit the student the luxury of leaving out multiplication signs and omitting parentheses around function arguments. But you, the author, are required to be more explicit in separating one name from another. Notice that "17angle" is fine - TUTOR will recognize this as meaning "17×angle". But "rangle" can't be pulled apart into "(r)(angle)" because you might have meant "(ran)(gle)".

Repeated operations: the iterative -do-

With very little effort we can make other pretty designs out of our unit "halfcirc". For example:

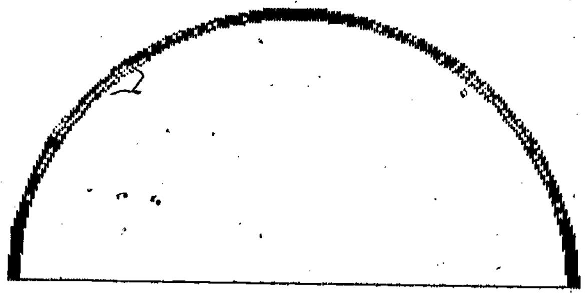


```
unit    stack
calc    x⇐256
        radius⇐7Ø
do      halfcirc,y⇐1ØØ,38Ø,7Ø
at      312
write   We used an
        iterative -do-.
```

The effect of the -do- statement is to set y to 1ØØ and do unit "halfcirc", then set y to 17Ø (the starting value of 1ØØ plus an increment of 7Ø) and do halfcirc again, and repeat until y reaches the final value of 38Ø. The format of the extremely useful iterative -do- statement is

```
do      unitname,index⇐start,end,increment
```

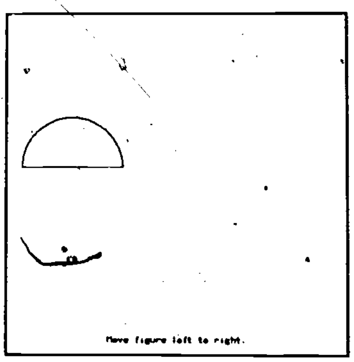
In the above example, the index "y" starts at 1ØØ and goes to 38Ø in increments of 7Ø. If no increment is specified, an increment of one is assumed: "do   halfcirc, radius⇐1Ø1,1Ø5" will make an arc five dots wide:

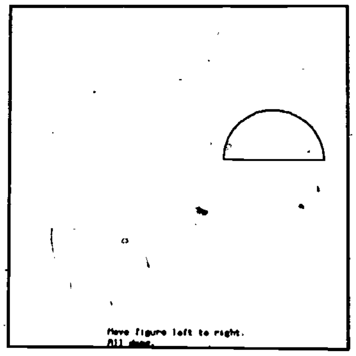


(Greatly enlarged.)

The iterative -do- statement also helps in making animations. The following statements will cause the half-circle to move horizontally across the screen:

```
unit    march
at      312Ø
write   Move figure left to right.
calc    y⇐28Ø
        radius⇐75
do      anim,x⇐1ØØ,35Ø,5Ø
do      halfcirc                    $$ draw final figure
at      322Ø
write   All done.
*
unit    anim
do      halfcirc                    $$ draw figure
catchup                             $$ wait for it to finish
pause   1                           $$ pause an additional second
mode    erase
do      halfcirc                    $$ erase the figure
mode    write
```


Move figure left to right.


Move figure left to right.
All done.

We simply -do- unit "anim" repeatedly for different values of x, the horizontal position of the figure on the screen. Unit "anim" does unit "halfcirc" twice, once to draw and once to erase the figure interrupted by a one-second pause. The -catchup- command insures that a second will elapse from the end of drawing the figure on the screen until the beginning of erasing it.

Now that you have studied -define-, -calc-, and -do-, you have learned the basic techniques of how to tell PLATO what calculations you want performed. We have applied these tools to a variety of display generation problems, and later we will use calculations for controlling sequencing in a lesson and for judging responses. Also, you have perhaps gained added insight into the value of a subroutine; look at how many different ways we have used that single unit "halfcirc"!

Showing the value of a variable

We have learned how to calculate and how to store results in variables. How do we show these results on the screen? If we perform:

```
calc    y⇐5sqrt(37)
```

(where "sqrt(37)" means "the square root of 37") how do we later show the value of y? Assume we have defined y. Perhaps we could use this:

```
write   y
```

No, that won't work; that will just put the letter "y" on the screen. The -write- command is basically a device for displaying non-varying text, not for showing the value contained in a variable. We need another command:

```
show    y
```

This will show the value of y in an appropriate format (-show- picks an appropriate number of significant figures and will use a scientific format such as 6.7×1Ø^{13} if the number is large enough to require it). By using -show- instead of -write-, you tell TUTOR that you want the stored value to be shown rather than just the characters in the tag.

The -show- command will normally choose 4 significant figures, so that a typical display might be "-23.47". You can specify a different value by giving a second "argument" (arguments are the individual pieces of the tag of a statement):

```
show    y,8           $$ 8 significant figures
```

The arguments of the -show- can, of course, be complicated expressions:

```
show    1Ø+3Øcos(2angle),format+2
```

In fact, it is a general rule that you can use complicated expressions anywhere in TUTOR statements: for example, "draw   5rad+225,34L;123-L^2,28L"!

Here is a short program which uses -show- to display a table of square roots of the integers from 1 to 15:

```
define  N=v1
unit    roots
at      31Ø            $$ write titles for the two columns
write   N
at      325
write   N^(1/2)
do      root,N⇐1,15
*
unit    root
at      41Ø+1ØØN
show    N
at      425+1ØØN
show    sqrt(N)
```

```
N        N^(1/2)

1        1
2        1.414
3        1.732
4        2
5        2.236
6        2.449
7        2.646
8        2.828
9        3
1Ø       3.162
11       3.317
12       3.464
13       3.6Ø6
14       3.742
15       3.873
```

The last statement could also be written as "show $N^{1/2}$". This technique of making tables, including the use of the -do- index (N) to position the displays (as in "at 425+100N") is an important and powerful tool.

There are other commands for displaying variables: -showe- (exponential), -showt- (tabular), -showa- (alphanumeric), -showo- (octal), and -showz- (show trailing zeroes). These are described in detail in reference material mentioned in Appendix A.

Although -write- is basically designed for non-variable text, combinations of text and variables occur so often that TUTOR makes it easy to "embed" a -show- command within a -write-:

```
        write   The area was <s,13.7w,6> square miles.
```

The embedded "s" indicates a -show- command and the remainder "13.7w,6" is its tag. Other permissible abbreviations include "o" (showo), "a", (showa), "e" (showe), "t" (showt) and "z" (showz). The above -write- statement is equivalent to

```
        write   The area was
        show    13.7w,6
        write    square miles.
```

Passing arguments to subroutines

When you write "show   13.7w,6", you are passing two pieces of information to the -show- command. You are giving two numerical "arguments" (13.7w and 6) to the TUTOR machinery that performs the -show- operations. Similarly, we created a half-circular arc with "circle radius,x,y,0,180" in which we passed five arguments to the TUTOR circle-making machinery. Sometimes certain arguments are optional: "show   13.7w" will use a default second argument of 4 (significant figures), and omitting the last two arguments in a -circle- command ("circle radius,x,y") will cause a full circle to be drawn rather than an arc. When we pass one argument to the -at- command ("at   1215"), we mean course grid; when we pass two arguments ("at   125,375"), we mean fine grid.

This notion of passing arguments to TUTOR commands, with some arguments optional, also applies to your own subroutines, such as unit "halfcirc". The "halfcirc" subroutine needs three arguments (radius, x, and y) to do its job.

We passed these arguments by assigning values to variables and letting "halfcirc" pick up those values and use them:

```
define  radius=v1,x=v2,y=v3
unit    vary
calc    radius⇐1ØØ
        x⇐15Ø
        y⇐3ØØ
do      halfcirc
calc    radius⇐5Ø
do      halfcirc
*
unit    halfcirc
circle  radius,x,y,Ø,18Ø
draw    x-radius,y;x+radius,y
```

Notice that the second -do- will use the original "x" and "y", since these variables have not been changed. It is as though we passed only one argument ("radius") to the subroutine.

TUTOR permits another way of writing this sequence which looks similar to the way one passes arguments to the "built-in subroutines" (-show-, -circle-, -at-, etc.):

```
define  radius=v1,x=v2,y=v3
unit    vary
do      halfcirc(1ØØ,15Ø,3ØØ)
do      halfcirc(5Ø)
*
unit    halfcirc(radius,x,y)
circle  radius,x,y,Ø,18Ø
draw    x-radius,y;x+radius,y
```

The statement "unit   halfcirc(radius,x,y)" tells TUTOR that when this unit is done as a subroutine, arguments are to be passed to it. The statement "do   halfcirc(1ØØ,15Ø,3ØØ)" tells TUTOR to pass the listed arguments to the "halfcirc" subroutine for its use. The arguments are passed in the order listed:

```
do      halfcirc(1ØØ,15Ø,3ØØ)
.                  |    |  |
.                  |1   |  | 3  (pass 3 arguments)
.                  |   2|  |
.                  v    v  v
unit    halfcirc(radius,x,y)
```

These variables are now set for use in the subroutine. It is precisely as though we had assigned values to "radius", "x", and "y" by using -calc-. If some arguments are omitted, these are not transferred:

```
        do      halfcirc(5Ø)
                    |                 (pass 1 argument)
                    v
        unit    halfcirc(radius,x,y)
```

In this case the variables "x" and "y" have not been assigned new values, so they retain the values they had. (These values are 15Ø and 3ØØ, but they could be different if there were -calc- statements in "halfcirc". For example, if we append "calc x⇐75" to the end of unit "halfcirc", "x" would now be 75, although it was 15Ø during the making of the first display, having been passed this value by the first -do-.)

Arguments to be passed need not be simple numbers. Each argument can be a complicated expression. The expressions are evaluated, then passed in order:

```
        do      halfcirc(3.4radius-25,radius+25y,2ØØ+y)
         .
         .                      1        2            3
         .
         .
        unit    halfcirc(radius,x,y)
```

It is as though we had written

```
        calc    arg1⇐3.4radius-25
                arg2⇐radius+25y
                arg3⇐2ØØ+y
                radius⇐arg1
                x⇐arg2
                y⇐arg3
```

Just as the -at- command handles its arguments differently depending on the number of arguments (one for coarse grid and two for fine grid), so it is possible for your subroutines to do such things. There is a TUTOR-defined "system variable" named "args" which always contains the number of arguments passed the last time a subroutine was done. By "system variable" we mean a variable separate from the student variables (v1 through v15Ø) whose contents are assigned by TUTOR rather than by you. You do not define system variables, they are already defined for you. (Indeed, if you say "define args=v3", you will override TUTOR's definition of the meaning of "args", so that "args" will mean "v3" rather than "the number of arguments passed to a subroutine".) In chapter VI (Conditional Commands) you will see how you could do different things in a subroutine conditional on the value of "args", similar to the kind of thing the -at- command does.

Our subroutine "halfcirc" uses three student variables: v1, v2, and v3, defined as "radius", "x", and "y". Another subroutine could use the same variables for carrying out its work, but it must be kept in mind that -do-ing this subroutine will affect v1, v2, and v3, since arguments will be passed.

Suppose one subroutine uses another, with "nested" -do-s like this:

```
•
•
•
do      A(5)            $$ v11⇐5
•           PASS
•
•
unit    A(v11)
do      B(3+v11)
calc    v11⇐1Øv11       $$ v11⇐5Ø
•
•               PASS
•
unit    B(v25)          $$ v25⇐8
```

Variable v11 ends up with the value 5Ø. It is advisable to use different variables in the two subroutines. Here unit A uses v11 and unit B uses v25. It can lead to confusion or even logical errors if B also uses v11 to do its work, since -do-ing B will affect the value of v11 used by A. Here is the structure to be avoided:

```
•
•
•
do      A(5)            $$ v11⇐5
•
•           PASS
•
unit    A(v11)
do      B(3+v11)
calc    v11⇐1Øv11       $$ v11⇐8Ø
•
•               PASS
•
unit    B(v11)          $$ v11⇐8
```

Now variable v11 ends up with the value 8Ø rather than 5Ø. This is due to the effect on v11 of the "do   B(3+v11)" statement, which assigns the value of 8 to v11 by passing the argument.

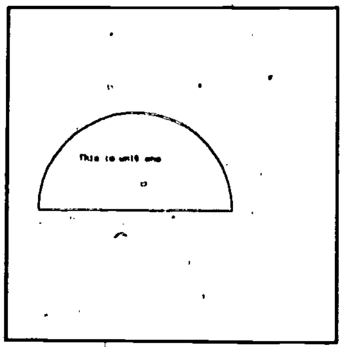
This concludes our discussion of calculations for now. We can calculate, save results, use them to make displays, and show the values. In the next section we will use calculations in association with guiding the sequencing of a lesson.

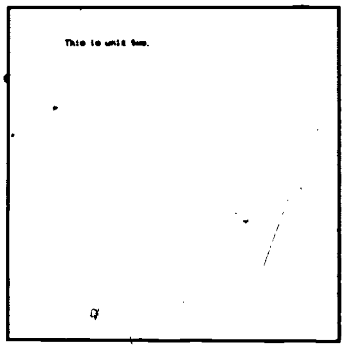
V. Sequencing of units within a lesson

We have discussed many units which make different kinds of displays. In some cases, the main units had other units attached to them by means of -do-. Upon completion of a main unit, the student can proceed to the next one by pressing NEXT. A greater variety of inter-unit connections is needed to build a complete lesson which includes optional help sequences, branches to remedial sections when the student is having trouble, an index that gives the student some control over the order of presentation, etc. In this section we will discuss in more detail how to build rich interconnections into a lesson. This discussion builds on the introduction to such matters presented in chapter I.

It is often desirable to skip over some units, particularly if they are used as subroutines, not as main presentation units. We have seen that this can be done by using a -next- command to name the main unit which is to follow:

```
unit    one
next    two
do      dispone
at      1515
write   This is unit one.
*
unit    dispone
calc    radius ⇐(x⇐y⇐2ØØ)-5Ø
do      halfcirc
*
unit    two
at      412
write   This is unit two.
```


This is unit one.


This is unit two.

When TUTOR begins "executing" the statements in unit "one", it starts out assuming that the next physical unit, unit "dispone", will be the next main unit. However, TUTOR encounters a "next two" statement which says, "No, make a note that unit "two" will be next, rather than the next physical unit". The "do dispone" is then executed, which involves drawing a figure. Finally, we write "This is unit one", which is at the end of unit "one". Nothing more will happen until the student presses the NEXT key, at which time TUTOR looks at its notes and finds that unit "two" comes next, whereupon it erases the screen and starts executing unit "two". Had we not inserted the -next- command, TUTOR would have gone on to unit "dispone" by default.

To put it another way, TUTOR has a pointer which tells which main unit should come next. At the beginning of a main unit TUTOR places zero in this pointer to indicate that the next physical unit should be next. If no -next- command is encountered, we reach the end of the unit with the pointer still zero, and, when the student presses NEXT, TUTOR will by default proceed to the next physical unit. On the other hand, if we encounter a -next- command anywhere in the unit, it will alter this pointer so that later, when the student presses NEXT, the pointer is non-zero and is pointing to whatever unit we have specified.

It should be clear from this discussion that the -next- command can be executed anywhere in the unit without changing its effect. Nevertheless, it is important to place the -next- command near the beginning of the unit. The advantage is that you can then see at a glance what is the main sequence flow. If the -next- command is buried far down in the unit, you have to hunt for this crucial information. You put such unit information at the beginning of a unit for the same reason that you define appropriate names for the variables you use: you or a colleague may have to read through the lesson months after it was written!

Here is another example which may illuminate the manner in which the -next- pointer is handled:

```
unit    silly
next    A
next    B
next    C
*
unit    sillier
```

Well, what unit will be next? Answer: unit "C"! The pointer starts out cleared to zero (which implies the next physical unit), then gets set to "A", then to "B", and finally to "C". Each succeeding -next- command overwrites what had previously been in the pointer.

It is also possible to clear the next pointer yourself by -next- with no tag or "next   q" ("q" for "quit specifying something"). Either of these forms will clear the next pointer so that the next physical unit will come next. In other words, the sequence

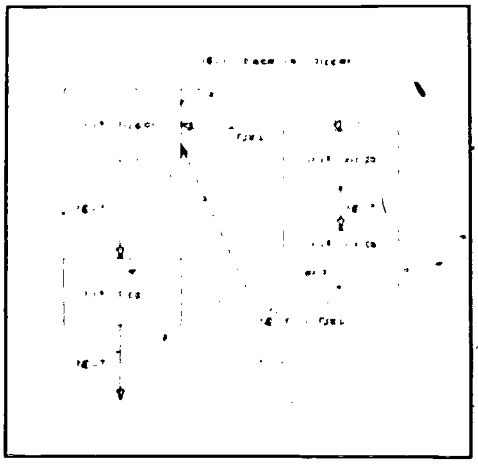
```
unit    start
next    silly
next    q         $$ or just "next" with no tag
*
unit    again
```

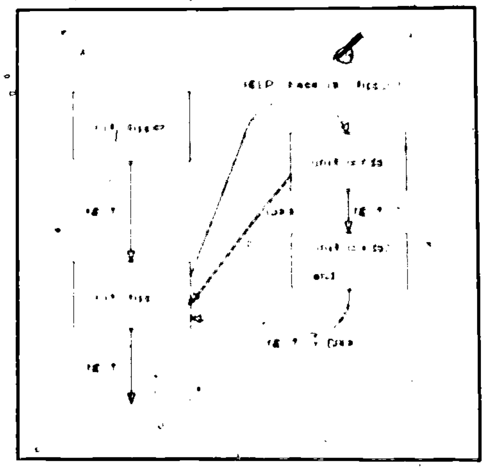
will proceed from unit "start" to unit "again" because the "next   q" cancels the "next   silly". Such seemingly meaningless manipulations are mentioned here for completeness and as aids to explaining how TUTOR handles a unit pointer such as that associated with the -next- command. Later these manipulations will make more sense. The important point is that you have complete control over the pointer. You can set it or clear it with an appropriate -next- command.

The existence of "next   q" (and related statements) means that "unit   q" is not a permitted statement: you are not allowed to name a unit "q" because the possible confusion. For similar reasons we will see later that a unit cannot be named "x".

Another use of pointers is in specifying optional "help" sequences which the student can request by pressing the HELP key. Such optional sequences are important tools in making the lesson cater to the needs of individual students of diverse backgrounds and abilities. Here is an example:

```
unit    dipper
help    words         $$ specify a help unit
at      1215
write   Today we will discuss Ursa Major.
*
unit    dippy
help    words         $$ specify a help unit
at      2213
write   Ursa Major is in the northern sky.
.
.
.
unit    words
at      1525
write   Ursa Major is the Latin name for the
        constellation which contains
        the "Big Dipper".
        (Press NEXT for more help,
        or Press BACK.)
*
unit    words2
at      1525
write   "Ursa" means "bear".
        "Major" means "bigger".
end
```





The -help- command is used to specify a "help" unit, which may be just the first unit in a long help sequence. If you provide help this way, the student can get it by pressing the HELP key. (Conversely, if there is no -help- command, the HELP key has no effect). When the student enters the help sequence, his screen is erased to clear the way for the display generated by the first help unit. The student may at any time press BACK or shift-BACK to return to "home base", the main unit he was in when he requested help. A "base" pointer retains the name of the "base unit" -- the unit to return to. In the example, if you press HELP in the base unit "dippy", TUTOR remembers "dippy" and jumps to "words", from where the BACK key will take you back to "dippy". If instead you press NEXT, you advance to "words2", where again you can press BACK or shift-BACK to return to "dippy". From "words2" you will also return to "dippy" upon pressing NEXT because the -end- command in unit "words2" signals the end of the help sequence.

It is almost as though the student had two screens he can look at! He starts the lesson in the first unit of a normal, non-help sequence and advances in this sequence until he requests help, at which point he turns his attention to a different, parallel sequence of units, almost as though he turned to use another terminal beside him. He can get back to the original sequence by pressing BACK, almost as though he turned back to the terminal in front of him. The usefulness of such a parallel sequence is not limited to help sequences but can be used to provide review, a desk calculator mode, a dictionary of terms, tables of data, etc., or for any situation in which the student temporarily needs a second terminal "off to the side".

It is possible to access yet another help sequence when you are already in a help sequence, but BACK will return you to the original base unit, not the help unit you were in when you requested the second help sequence. This is due to the fact that there is only one base pointer, which is not changed by the second help request. If there is already a base unit specification, TUTOR does not alter it.

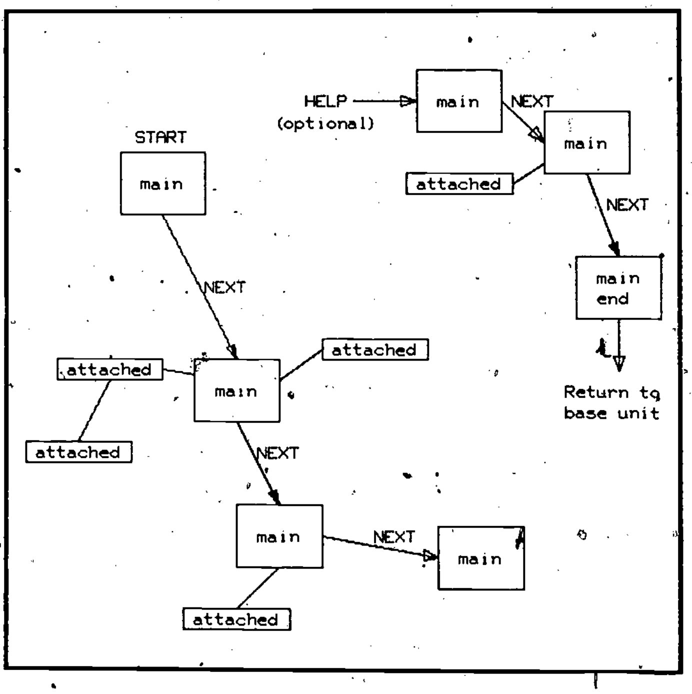
You can alter the base unit pointer yourself with a -base- command. If you put a -base- command with no tag in unit "words" you will prevent a return to "dipper" or "dippy". The -base- command with no tag or a "base q" statement clears the base pointer so that TUTOR forgets there was any place to return to and thinks that you are not in a help sequence. (You should notice that the -end- command in unit "words2" is now ignored; the -end- command has no effect in a non-help sequence.) This -base- (blank or "q" tag) is used quite often, for it is frequently convenient to put the student into a non-help sequence even though he reached a certain point by pressing HELP. TUTOR automatically clears the base pointer whenever the student reaches the corresponding base unit, by whatever means.

You can change the base pointer to point to some unit other than the original one. Imagine that we place in unit "words" the statement

```
        base    dispone
```

This means TUTOR will eventually return to "dispone" rather than "dipper" or "dippy". This is occasionally a useful technique. For example, you might like to return to a unit just ahead of the original one in order to ease back into the original context. Notice, too, that while -base- with no tag (or "q") can change a help sequence into a non-help sequence, so "base unitname" can change a non-help sequence into a help sequence by naming a unit to return to.

You probably will not need all of the features of -help-, -base-, and -end- described above, but hopefully the discussion has clarified how they do their work. We have also picked up some terms which will be quite useful in later discussions: we can now talk about "non-help sequences" of "main units" and "help sequences" of "main units". It should also be pointed out that a base unit may have other (auxiliary) units attached to it by -do-; and, of course, we return to the base unit itself, not to one of these attached units, even if the -help- command is located in an attached unit. More generally, a lesson may be thought of as a collection of main units which have attached units, and the student moves from one main unit to another. He may enter a help sequence of main units, each of which may -do- attached units. While he is in the help sequence, TUTOR remembers which main unit is the "base" unit to return to when -end- is encountered or when BACK or shift-BACK is pressed.



You may have realized that -help- and -base- are quite similar to -next- in that all three commands set pointers (which have different uses, however). In particular, if we say

```
        unit    lotshelp
        help    a
        help    b
        help    c
```

then the last one wins -- the help pointer ends up pointing at unit "c". We saw earlier that -next- works this way. Similarly, "help q" or -help- with no tag will clear the help pointer, thus making the HELP key inoperative.

You may find it helpful to think of a help sequence as a "slow" subroutine. Whereas a -do- command takes us to a unit and right back again, -help- makes possible an optional jump to a unit or to a sequence of units where the student may study for many minutes before returning to the base unit. Aside from the "slowness" and the necessity of pressing keys to go and return, there is one fundamental difference from a -do- situation: we return from help to the beginning of the base unit and re-execute the statements in the unit to restore the original display, whereas the return from a -do- is to the statement following the -do-.

This last point is sufficiently important to warrant an example:

```
unit    initial
at      2513
write   Set "a" to Ø.
calc    a⇐Ø
*
unit    repeat
help    trivial
at      2715
write   Increment "a" to ⊲s,a⇐a+1⊳ .
*
unit    trivial
at      312
write   Press NEXT or BACK.
end
```

(Of course, "a" must be defined.) If while in unit "repeat" we repeatedly press HELP, then BACK, we will repeatedly increment variable "a"; it increases by one on every return from the help sequence because the return is to the beginning of the base unit, and all the statements in unit "repeat" are re-executed. This is necessary to restore to the screen the display associated with unit "repeat", since the entire screen is erased when the HELP and BACK keys are pressed.

This example brings up a fundamental programming point: the question of "initialization". We might use such a structure for counting the number of times the student presses the HELP key (although we would then probably put the "a⇐a+1" in the help unit). In order to count something (requests for help, number of wrong answers, etc.), it is necessary to "initialize" the counting variable to zero before starting the process, and this initialization must precede and be outside the process itself. This can perhaps best be seen by moving the statement "calc a⇐∅" from unit "initial" to the beginning of unit "repeat":

```
.
.
.
unit    repeat
help    trivial
calc    a⇐∅
at      2715
write   Increment "a" to ⊲s,a⇐a+1⊳.
.
.
.
```

Imagine pressing HELP (and BACK) repeatedly. Now there will never be a change in the displayed value of "a", because on each return from the help unit "a" is again reset to zero, whereas previously that was done only within unit "initial".

We will encounter the question of initialization again and again in various guises. We did not mention these matters earlier partly because the iterative -do- command had the initialization built-in:

```
do      zonk,i⇐5,13
```

means "initialize 'i' to 5 and do 'zonk', then repeat by incrementing 'i' by one until it reaches 13".

Let us hasten to say that initialization questions are, of course, not unique to programming. The principal and interest due monthly on your car or house loan depend on the initial conditions of the loan. When you make fudge, you start with certain ingredients in the mixing bowl (the initial condition) and then you beat the mixture 2∅∅ times. You would no more restart with new, unmixed ingredients after each beating stroke than you would reinitialize a count of student errors after each attempt. In other words, questions of initialization are mainly questions of common sense, and we will make explicit comments about these matters only where confusion is likely. In the case of a return from a help sequence, you might have thought that TUTOR remembers the entire display originally made by the base unit, but, as we have seen, it must re-create the display by re-executing the commands in the base unit, which has side effects related to initialization questions.

Now let's move the "calc a⇐Ø" back to unit "initial" and modify the unit to look like this:

```
        unit    initial
        calc    a⇐Ø
        jump    repeat      $$ do not wait for the NEXT key
        *
```

The -jump- command acts much like the student pressing NEXT: the screen is erased and we move to a new main unit. The -jump- command is particularly useful in association with initializations, as in this example, where it is necessary to separate initializations from a process in a different unit. It would be superfluous to show the student a blank screen and to make the student press NEXT. Indeed, it should be a basic rule to minimize unnecessary keypresses so as not to frustrate the student. Notice that -jump- is <u>immediate</u> (like -do- and unlike the -next- or -help- commands) and that statements that follow -jump- in a unit <u>will not be executed</u> (unlike -do-, -next-, and -help-).

The base pointer is not affected by a -jump-: it remains zero if we are not in a help sequence, and it retains its base unit specification if we are in a help sequence. The -jump- simply takes us from one new main unit to another without having to press NEXT. Since it starts a new main unit, a -jump- cancels any -do-s which have been encountered; there will be no return from those -do-s.

When moving from one main unit to another, by -jump- or by pressing NEXT, the entire screen is erased unless the <u>first</u> of these two main units contains an "inhibit erase" statement.

Since -jump- takes the student from one main unit to another without altering the base pointer, it is possible to take a student to a help sequence immediately without his pressing HELP:

```
        unit    model
        .
        .
        .
        base    model
        jump    modhelp
        .
        .
        .
```

Initially, the base pointer is zero because we are in a non-help sequence. Then a -base- command is used to set the base pointer to unit "model" (the main unit we are presently in). The -jump- takes us to unit "modhelp". Now we are in a help sequence because the base pointer has been set. The return from the help sequence will be to the beginning of unit "model". Note the difference between "base model" and "base q" in unit "model": a "base q" statement would clear the already-cleared base pointer, whereas "base model" sets it to "model".

Summary of sequencing commands

You have learned many commands which enable you to control the sequencing of units in a lesson. These include commands which set pointers (-next-, -help-, -base-, etc.) and a couple of immediate branching commands (-do- and -jump-). You have seen how to have two parallel sequences of main units -- a non-help sequence and a help sequence -- and have used the -end- command to terminate a help sequence. Additional aspects of the connections among units will be discussed in chapter VI in the section on the -goto- command. We recall that the LAB, DATA, and BACK keys are activated by -lab-, -data-, and -back- commands, just as the HELP key is activated by the -help- command. The shifted HELP, LAB, DATA, NEXT, and BACK keys (abbreviated as HELP1, LAB1, DATA1, NEXT1, and BACK1) are activated by the commands -help1-, -lab1-, -data1-, -next1-, and -back1-. (When in a help sequence the BACK or BACK1 keys will cause a return to the base unit unless there are explicit -back- or -back1- commands to alter this.) Here is a unit which uses many of these commands:

```
unit    central
help    uhelp
help1   index
lab     simulate
lab1    calc
data    data
data1   news
at      1314
write   Press   HELP for assistance,
                shift-HELP for an index,
                LAB for simulation,
                shift-LAB for a calculator,
                DATA for tables of data,
                shift-DATA for class news.
```

This is an extreme case, but this unit gives the student six choices of help sequences, and which help sequence is entered depends on which key the student presses. In any of these cases the eventual return will be to this base unit. The commands -next-, next1-, -back-, and -back1- are somewhat different in that these do not cause a help sequence to be initiated: pressing the corresponding key does not alter the base pointer.

The same conventions apply to all these commands. In particular, a blank tag (or "q") disables the corresponding key by clearing the associated pointer. A non-help sequence can be changed into a help sequence by specifying a unit to return to with a "base unit" statement. A help sequence becomes a non-help sequence if we encounter a "base q" or "base" statement, since these clear the base pointer.

It is important to point out that all the unit pointers other than "base" are cleared when we start a new main unit (either by -jump- or by pressing a key such as NEXT, BACK, or HELP).

Notice that -jump- and -do- are basically author-controlled branching commands, while -help-, -back-, -data-, etc., permit the student to control the lesson sequence.

There is another way to enter a help sequence which is particularly useful in offering to the student an index to the various parts of the lesson. Suppose the lesson is organized into chapters or topics and you wish to let the student choose his own sequence. In particular, he can skip ahead, go back, or review material. It is desirable that he be able to go to an index or table of contents at any time. One way to provide access to the index is to put a "data   table" statement in every main unit. Then the student can hit the DATA key and jump to unit "table" at any time. Unit "table" would contain a list of topics for the student to choose from, and it should contain a "base" statement to insure that the chosen topic be entered as a base sequence. Another way to provide this kind of index is by means of a single -term- command:

```
unit    table
base
term    index
at      1218
write   Choose a chapter:
           a)  Introduction
           b)  Nouns
           c)  Pronouns
           d)  Verbs
arrow   1822
answer  a
jump    intro
answer  b
jump    unoun
answer  c
jump    pron
answer  d
jump    verb
```

The presence of "term   index" in the unit "table" makes it possible for the student at any time to press the TERM key and type "index" to reach unit "table". (The TERM key is the shifted ANS key on the keyboard.) When he presses TERM, TUTOR responds by asking him at the bottom of the screen "what term?", whereupon he would type "index". He then reaches unit "table", where he can choose a chapter. You can see that -term- is complementary to -help-: -help- in a main unit specifies where to go if HELP is pressed while in that main unit, whereas the presence of -term- in a unit specifies that the unit can be entered from anywhere in the lesson. It is an error to put another -term- command with the same tag in another unit, for then TUTOR doesn't know which unit to enter.

The name -term- stems from an early use of this kind of facility to provide access to a dictionary of "terms" -- special vocabulary used in a lesson. In such an application there are as many help units as there are terms to be defined, and each unit has an appropriate -term- command:

```
          unit     cardinfo
          term     cardiac
          at       1907
          write    "cardiac" means "pertaining to the heart".
          end
```

Except for situations of this kind, it is strongly recommended that you limit yourself to having only one unit with a -term- in it, and its tag be "index". This greatly simplifies the instructions to the student on how to use the lesson and reduces to a minimum what he must remember in order to move around in the lesson. In the index unit you describe the various options that are available. Even for providing a dictionary of terms, this scheme is probably preferable -- one of the options could be "dictionary of terms", which in turn would show a list of the words whose definitions are available.

It is possible to have additional -term- commands in the unit to provide synonyms:

```
          unit     table
          base
          term     index
          term     contents
          term     choice
          at       1218
          write    Choose a topic...
```

These additions insure that the student will reach this unit by TERM-index, or TERM-contents, or TERM-choice.

The -imain- command

An alternative to "TERM-index" is to tell the student to press a key such as LAB to reach an index page. If this index is in unit "table", you must then put the statement "lab   table" in every main unit, since all unit pointers are cleared when a new main unit is entered. A better way to do this is to use an -imain- command which specifies a unit to be done initially in every main unit:

```
imain   setit                              .
.                                          .
.                                          .
.                                          .
unit    a                                  unit    a
.                                          do      setit
.                                          .
.                   IS                     .
unit    b       EQUIVALENT                 unit    b
.                   TO                     do      setit
.                                          .
.                                          .
unit    c                                  unit    c
.                                          do      setit
.                                          .
.                                          .
unit    setit                              unit    setit
lab     table                              lab     table
.                                          .
.                                          .
.                                          .
```

The -imain- command names unit "setit" to be done at the beginning of every main unit.

You can specify all kinds of initializations to be performed in each main unit. For example, you might advertise the LAB key with this display at the bottom of the screen:

$$\boxed{\text{Press LAB for an index}}$$

In this case you would write something like

```
imain   setit
.
.
.
unit    setit
lab     table
at      3220
write   Press LAB for index
draw    3120;3144;3244;3220;3120
```

Now the display will appear with each main unit, and the LAB key will be activated.

The -imain- command sets a pointer, just as the -help- and -base- commands do. You can change the associated unit by executing another -imain- command:

```
        imain   setit
        .
        .
        .
        imain   other
        .
        .
        .
```

You can stop having an imain-associated unit done by using "imain   q" or "imain" (blank tag) to clear the -imain- pointer.

While any key may be used to access an index, many authors have agreed to use shift-DATA, in order to provide some uniformity from one lesson to another. This reduces the number of new conventions a student must learn when studying new material.

There is a similar -iarrow- command which can be used to specify a unit to be performed every time a student enters a response. If the -iarrow- command is itself located in the -imain- unit, all -arrow-s will be affected.

VI. Conditional commands

It is important to be able to specify the sequencing of a lesson conditionally. We would like to jump past some material on the condition that the student has demonstrated mastery of the concept and needs no further practice. Or we would like to take the student to a remedial sequence conditionally, the condition being poor performance on the present topic. Or which help sequence we offer might be conditional on the number of times help has been requested. All of these examples imply a need for conditional sequencing or branching statements, where the condition may be specified by calculations involving the status of the student.

The usefulness of conditional branching is not limited to the sequencing of major lesson segments, but extends to many calculational or display situations. For example, we might need to -do- conditionally one of several possible subroutines in the course of presenting a complex display to the student. This chapter will show you how to perform these and similar conditional operations.

Here is an example involving a conditional -do- statement:

```
        unit    setup
        calc    N⇐-1
        jump    home
        *
        unit    home
        next    home
        at      2Ø1Ø
        do      N,neg,uzero,One,utwo
        at      1215
        write   N equals  ◁s,N▷.
        calc    N⇐N+1
        *
        unit    neg
        write   Unit "neg".
        *
        unit    uzero
        draw    ;21Ø,26Ø;2Ø6Ø;2Ø1Ø
        *
        unit    One
        circleb 2ØØ,25Ø,25Ø,Ø,27Ø
        *
        unit    utwo
        write   Unit "two".
```

The new element is the conditional -do- statement in unit "home". If N is negative, that statement is equivalent to "do   neg". If N is zero, the statement is equivalent to "do   uzero", and so on:

```
          do       N,neg,uzero,One,utwo
```

is equivalent to

```
          do       neg.        if N is negative
          do       uzero       if N is zero
          do       One         if N is 1
          do       utwo        if N is 2 or greater
```

Note that unit "utwo" will come up repeatedly because it is the last unit named in the conditional -do- statement. The list of unit names can be up to 1ØØ long:

```
          do       N,neg,uzero,One,utwo,dispone,
                   zon,zip,figure,ultima
```

If N is 7 or greater, this statement is equivalent to "do   ultima".

The "conditional expression" (N in this case) can be anything: it can be as complicated as "3x - 5 sqrt(N)" and can even involve assignments as in "N⇐35-x". The value of the expression is rounded to the nearest integer before choosing a unit from the list of units; and if the rounded value is negative, the first unit in the list is chosen. For example, if the expression is -.4, it rounds to zero, in which case the second unit in the list is chosen.

In a conditional -do- each unit named may involve the passing of arguments:

```
          do       3N-4,circ(25,75),box(45),x,flag,circ(1Ø,3Ø)
                            neg          Ø     1   2     ≥3
```

So far we have encountered the following sequencing commands: -do-, -jump-, -next-, -next1-, -back-, -back1-, -help-, -help1-, -lab-, -lab1-, -nextnow-, -data-, -data1-, and -base-. When the tag of such a command is just a single unit name (e.g., in a statement like "help   uhelper"), we say it is "unconditional". To make a "conditional" statement out of any of these, we follow the same rule -- state the conditional expression followed by a list of unit names. So we might have

```
          data     N-5,zonk,q,zap,zing,x
            expression  negative  zero  one  two  three or greater
```

Here "q" has the same meaning it had in unconditional pointer-associated statements: the "data" pointer is cleared so that the DATA key is disabled. This can be used to cancel the effect of an earlier -data- command in this main unit. (Remember that all the unit pointers are cleared when we start a new main unit.) The unit name "x" has the special meaning "don't do anything!" In the example shown, if the condition (N-5) is three or greater, this -data- command has no effect at all: we "fall through" to the next statement without affecting the "data" pointer. Similarly, if a unit name in the conditional -do- discussed above is replaced by "x", no unit will be done for the corresponding condition: we "fall through" to the next statement.

This "x" option is extraordinarily useful. Consider the following situation:

```
        jump    correct-5,x,done

        (then show the next item)
```

If (correct-5) is negative (i.e., the student has made fewer than 5 correct answers), we "fall through" to the presentation of the next item. If, however, he has 5 or more correct, the condition (correct-5) will be zero or greater and we jump to unit "done".

Logical expressions

The last example can be written in an alternative form which improves the readability:

```
        jump    correct<5,x,done
```

This says "fall through if correct is less than 5, otherwise jump to done". The condition (correct<5) we call a "logical expression" because it has only two possible values, "true" (-1) or "false" (Ø), whereas numerical expressions can have any numerical value. Since a logical expression can have only two values (-1 if true or Ø if false) it is pointless to list more than two unit names after the condition.

Actually, because of rounding, the form "jump   N<5,x,done" is more precise than the form "jump   N-5,x,done". Suppose that N is 4.8. Then "N<5" is true (-1), which rounds to -1, which implies "x". But N-5" is -Ø.2, which rounds to zero, which implies "done". Such differences show up whenever some of the variables need not have integer values.

Here is another example:

```
        do      c-b,far,near,far
```

will do unit "near" if c and b differ by no more than Ø.5, since in that case "c-b" will lie between -Ø.5 and +Ø.5, which rounds to zero.

On the other hand

```
          do        c=b,same,diff
```

will do unit "same" only if c and b are equal, not just nearly the same. The condition "c=b" is true (-1) only if c is precisely equal to b.

There are six basic logical operators: =, ≠, <, >, ≤, and ≥, which mean equal, not equal, less than, greater than, less than or equal, and greater than or equal. "do a≠b,diff,same" is equivalent to "do a=b,same,diff". The operators involving equality (=, ≠, ≤, and ≥) consider two numbers to be equal if they differ by less than one part in $1Ø^{11}$ (relative tolerance) or by an absolute difference of $1Ø^{-9}$, whichever is larger. This is done to compensate for small roundoff errors inherent to computers due to their very high but not infinite precision. One consequence is that all numbers within $1Ø^{-9}$ of zero are considered equal by these logical operators.

You can mix logical expressions with numerical expressions in fruitful ways:

```
          calc      x⇐1ØØ-25(y>13)
```

gives "x⇐125" if y is greater than 13 ("y>13" if true is -1) or it gives "x⇐1ØØ" if y is less than or equal to 13 ("y>13" if false is Ø). To clarify this, suppose that y is 18 or y is 4:

<u>y=18</u>	<u>y=4</u>
1ØØ-25(y>13)	1ØØ-25(y>13)
1ØØ-25(18>13)	1ØØ-25(4>13)
1ØØ-25(-1)	1ØØ-25(Ø)
1ØØ+25	1ØØ-(Ø)
125	1ØØ

In these applications it would be nice if "true" were +1 rather than -1, but the much more common use of logical expressions in conditional branching commands dictates the choice of -1, since the first unit listed is chosen if the condition is negative.

You can combine logical expressions:

```
          [(3<b) $and$ (b<5)]
```

is true (-1) only if <u>both</u> conditions (3<b) and (b<5) are true. In other words, b must lie between 3 and 5 for this expression to have the value -1. Similarly,

```
          (y>x) $or$ (b=2)
```

will be true if <u>either</u> (y>x) is true <u>or</u> (b=2) is true (or both are true).

Finally, you can "invert" the truth of an expression:

```
not(b=3c)
```

is true if (b=3c) is not true. This complete expression is equivalent to "b≠3c".

The combining operations and, or, and "not" make sense only when used in association with logical expressions (which are -1 or Ø). For instance, [b>c and 19] is meaningless and will give unpredictable results. (If you have done a great deal of programming, you might wonder about special bit manipulations, but there are separate operators for masking, union, and shift operations, as discussed in chapter IX.)

The conditional -write- command (-writec-)

A very common situation is that of needing to write one of several possible messages on the screen. For example, you might like to pick one of five congratulatory messages to write after receiving a correct response from the student:

```
unit     congrat
randu    N,5               $$ let TUTOR pick an integer from 1 to 5
at       1215
do       N-2,ok1,ok2,ok3,ok4,ok5
*
unit     ok1
write    Good!
*
unit     ok2
write    Excellent!
*
unit     ok3
write    I'm proud of you.
*
unit     ok4
write    Hurray!
*
unit     ok5
write    Great!
```

The -randu- command, "random on a uniform distribution", tells TUTOR to pick an integer between 1 and 5 and put it in N. We then use this value of N to do one of five units to write one of five messages. There is a much more compact way of writing this:

```
unit     congrat
randu    N,5
at       1215
writec   N-2,Good!,Excellent!,
         I'm proud of you.,
         Hurray!,Great!,
```

The -writec- command is similar to that of a conditional branching command, but the listed elements are pieces of text rather than unit names. Because -write- can be used to display any kind of text (including commas), it is necessary to use a different command name (-writec-) to indicate the conditional form of -write-, whereas in branching statements the commas separating the unit names are enough to tell TUTOR that it is a conditional rather than unconditional form.

You can write whole paragraphs with nice left margins, just as with the -write- command:

```
writec  N,,,Good!,Excellent!,
        I'm proud of
        you and so
        is your mother.,
        Hurray!,Great!,
```

The elements of text are set off by commas. If N is 3, the student will see a three-line paragraph, since there are no commas at the end of "of" and "so". If N is -1 or Ø, no text will be displayed, since there is no text between the first few commas. Note that "x" is not the fall-through that is for a unit name in a conditional branching command: here "x" is a legitimate piece of text which can be displayed, so the ",," form is the "fall-through".

If you want commas to appear in some of your text elements, you have a problem, since the commas delimit elements:

```
writec  N,Hello!,How are you, Bill?,Hi there!,
```

If N is zero, we will see "How are you", not "How are you, Bill?" The solution is to use a special character (‡):

```
writec  N‡Hello!‡How are you, Bill?‡Hi there!‡
```

Now if N=Ø we will see "How are you, Bill?". While this special character (‡) is <u>required</u> if text elements contain commas, you may prefer to use it always, even when there are no commas.

The same kinds of embedding of other commands permitted by -write- are permitted with -writec-:

```
writec  2c=b,I have ◁s,ap▷ apples.,
        I will buy ◁s,peachy▷ peaches.,
```

The -writec- is affected by -size- and -rotate- commands, just like -write-.

The conditional -calc- commands: -calcc- and -calcs-

The effects of -writec- can be achieved by a conditional -do- and a bunch of units containing the text elements, but we have seen that this is a clumsy way to do it. We would often like to calculate one of several things based on a condition. This, too, could be done with a conditional -do- to one of several units containing the calculations, but this is cumbersome. We saw one shortcut already:

```
        calc    x⇐1ØØ-25(y>13)
```

is equivalent to "x⇐125" if y>13 and to "x⇐1ØØ" if not. This can also be written as

```
        calcc   y>13,x⇐125,x⇐1ØØ
```

The -calcc- is strictly analogous to -writec-. It indicates a list of calculations to be performed, dependent on a condition. The elements in the list are calculations rather than pieces of text or unit names.

Very often each of the calculations in the list consists of assigning a value to the <u>same</u> variable. In the example above both calculations assign a value to the variable "x". An even shorter way to write this kind of thing is

```
        calcs   N-5y,bin⇐37,5.2,y³+2,,2/N
```

This will store (-calcs-) one of five values in "bin", depending on the condition "N-5y". Note that if "N-5y" rounds to two, we do nothing: two commas in a row (,,) indicate "do nothing" in -calcs-, -calcc-, and -writec-.

The conditional -mode- command

For completeness it should be mentioned that the -mode- command also can be made conditional:

```
        mode    count-3,write,x,rewrite,erase,write
```

Here the list of elements following the condition is similar to the list of unit names in a -help- command: they are the names of the various possible screen display modes. The "x" option means "do nothing --- do not change the present mode".

The -goto- command

The -goto- command is a very mild version of the -jump- command. It does not initiate a new main unit and does not perform the initializations associated with starting a main unit: the screen is not erased, the help and other unit pointers are not cleared, and how deep we are in "do" levels is unaffected. It is most often used in its conditional form so we waited until this chapter to introduce it.

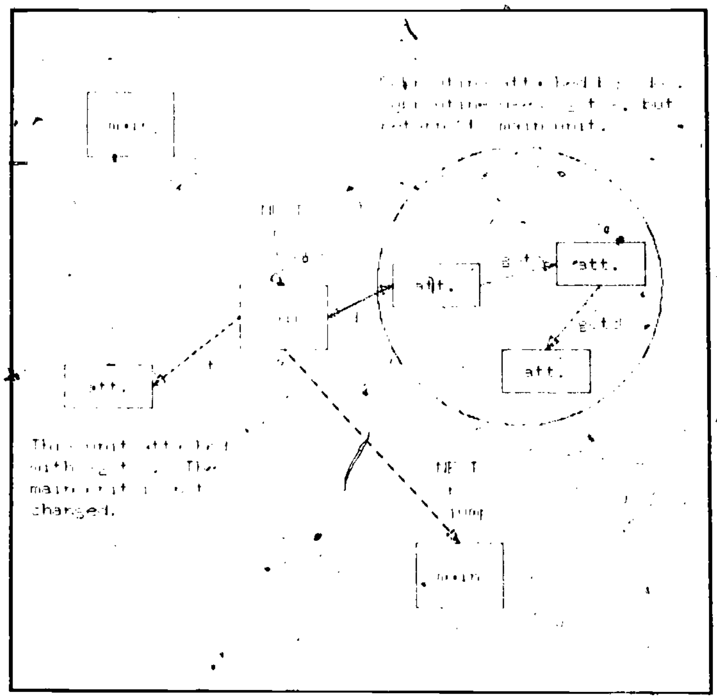
One common use of the -goto- command is to "cutoff" a unit prematurely:

```
unit    A
at      1315
write   You have now finished the quiz.
goto    score<90,fair,x
size    4
at      22ø5
write   Congratulations!
size    ø
*
unit    B
at      1912
write   The next topic is . . . . . .
.
.
.
unit    fair
at      1815
write   Your score was below 9ø.
*
unit    blah
.
.
.
```

In this example a score of 9ø or better will mean that we fall through the -goto- to display the large-size "Congratulations!" A score of less than 9ø will take us to unit "fair" to add "Your score was below 9ø" to the "You have finished the quiz" already on the screen. The -goto- does not erase the screen, nor does it change the fact that the main unit is still "A". When the student presses NEXT, he proceeds to unit "B", the main unit following unit "A". He does <u>not</u> proceed to unit "blah"..

Like -do-, the -goto- command attaches a unit without changing which unit is "home", whereas -jump- changes the main unit and performs the many initializations associated with entering a new main unit (full-screen erase, clearing the help pointers, forgetting any -do-s, etc.) As to the difference between -goto- and -do-, the main difference is that the -do- will normally come back upon completion of the attached unit, whereas -goto- does not come back: statements following the -goto- are normally not executed.

The relationships among main units and attached units and among -jump-, -goto-, and -do- may be clearer if you think of a lesson as being made up of a number of nodes or clusters each consisting of a main unit and its attached units:



Movement between main units is made by pressing NEXT (or HELP, BACK, etc.) or by executing a -jump-. These main units may form a normal sequence or a help sequence (see Chapter V). The -goto- and -do- commands attach auxiliary units to these main units.

Notice that completion of a unit reached by one or more -goto-s will cause TUTOR to "undo" one level if one or more -do-s had intervened in reaching this unit. The reason this occurs is that whenever TUTOR encounters a -unit- command (which terminates the preceding unit) TUTOR asks "Are we at the main-unit level?" If so, we have completed processing; if not, we must "undo" to the statement immediately following the last -do- encountered. This point deserves an example for illustration:

```
        unit    calcit
        do      sum
        show    total
        .
        .
        .
        unit    sum
        calc    total⇐Ø          $$ initialize "total"
        goto    addup
        *
        unit    addup
        .
        .                        $$ a calculation of "total"
        .
        unit    other
```

In unit "calcit" we -do- "sum", which initializes "total" and does a -goto- to unit "addup", where some kind of calculation is performed. When we run out of work (by encountering a -unit- command at the end of unit "addup"), TUTOR asks whether there was a -do-. There was a -do-, so control passes to the statement following the last -do-, which is "show total". All of this is perfectly reasonable and useful, but it should be pointed out that this property of the -goto-, that it preserves the required information to permit "undoing", has an odd side-effect. The presence of a -goto- in a done unit causes an exception (the only exception) to the description of -do- as a text-insertion device. Except for this case, the effect of a -do- is equivalent to inserting all the statements contained in the done unit in place of the -do- statement. But suppose we replace our -do- with the statements contained in unit "sum". We would have:

```
        unit    calcit
        calc    total⇐Ø} in place of "do      sum"
        goto    addup
        show    total
        *
        unit    addup
        .
        .
        unit    other
```

Now the -goto- cuts off the rest of unit "calcit", and the -show- will not be performed, in contrast with the case where we used a -do-. So the presence of a -goto- in a done unit causes a (useful) exception to the text-insertion nature of -do-.

Here is a summary of the basic properties of the -goto- command:

1) -goto- may be used to attach units with none of the initializations associated with -jump-;
2) statements which follow the -goto- will not be executed (like -jump- and unlike -do-);
3) a -goto- in a done unit does not cut off statements following the original -do- statement, which is an exception to the normal text-insertion nature of -do-.

Additional aspects of -goto- are discussed in Chapter VII (Judging Student Responses).

It is often convenient to cut off a unit with a -goto- of the form shown in this example:

```
unit    cuts
goto    expression,x,zonk,empty,x,empty
write   We fell through...
.
.
.
unit    empty
*
unit    zonk
.
.
.
```

Note that unit "empty" has nothing in it but serves merely to have a place to go to in order to cut off the end of unit "cuts". This is such a common situation that TUTOR provides an empty unit named "q" (for quit); the previous -goto- can be written as

```
goto    expression,x,zonk,q,x,q
```

and "goto   q" means go to an empty unit. The special meaning of q here makes it illegal to have your own unit named "q", just as it is not possible to name a unit "x". The use of "q" in a -goto- statement is somewhat different from the use of "q" in a -help- statement. You will recall from Chapter V that "help   q" means to quit specifying a help unit (by clearing the -help- pointer). Since "do   empty" can be rendered by the equivalent "do   x", the statement "do   q" (or a conditional form) is given the special interpretation of acting like a "goto   q".

The -goto- can be used in association with the -entry- command to skip over statements:

```
        .
        .
        .
        calc    b⇐∅
        goto    3f>5,leavit,x
        calc    b⇐f/2
                f⇐∅
        entry   leavit
        .
        .
        .
```

If 3f is greater than 5, we skip over intervening statements to entry "leavit". The -entry- command is equivalent to a special -goto- plus a -unit-:

```
        .
        .
        .
        {special goto    leavit}
        {unit            leavit} equivalent to (entry leavit)
        .
        .
        .
```

so that, unlike a -unit- command, -entry- does not terminate a unit but merely provides a named place to branch to. Its equivalence to a special hidden -goto- followed by a -unit- command means that an entry is completely equivalent to a unit except for not terminating the preceding statements. For this reason it is possible to use an entry name with -do-, -jump-, -help-, etc.

The conditional -goto- is often used for repetitive operations similar to those carried out with -do-. Here are two versions of a subroutine to add the cubes of the first ten integers:

```
        -do-                          -goto-

        unit    add                   unit    add
        calc    total⇐∅               calc    i⇐1
        do      add2,i⇐1,1∅                   total⇐∅
        *                             goto    add2
        unit    add2                  *
        calc    total⇐total+i³        unit    add2
                                      calc    total⇐total+i³
                                              i⇐i+1
                                      goto    i≤1∅,add2,x
```

The last two statements in the -goto- example could be combined into one: goto (i⇐i+1)≤1∅,add2,x. For the simple task of adding ten numbers, the -do- form is certainly easier to construct, but situations occasionally arise where it is easier to construct a repetitive loop using a conditional -goto-.

Except for not changing how many levels deep in -do-s we are, -goto- is quite similar to -do-. Although the feature is little used, it is even possible to pass arguments to a subroutine with a -goto-: "goto zonk(12,25)". Arguments may also be passed in a conditional -goto-: "goto 3N-4,alpha(2+count),x,beta(15,2N),q".

The conditional iterative -do-

The conditional and iterative -do- can be combined so that, on each iteration, the conditional expression selects which unit to do this time:

```
        do      N+3,ua,ub,uc,ud,i⇐1,12
                    neg  ∅   1  ≥2
```

For each value of i (from 1 to 12), the expression "N+3" is evaluated, which determines which subroutine will be done. For example, if "N+3" is ∅, the above statement is equivalent to "do ub,i⇐1,12". Usually a conditional iterative -do- is used in situations where the conditional expression ("N+3") is not changing, but doing one of the subroutines can change N so that a different subroutine is used on the next iteration. A more straightforward example of such manipulations is this:

```
        do      i-2,ua,ub,uc,ud,i⇐1,4
```

For i equal to 1, the condition "i-2" is -1, so we do "ua". Then i is incremented to 2, and we do "ub", etc. This statement is, therefore, equivalent to the sequence

```
        do      ua
        do      ub
        do      uc
        do      ud
```

As usual, the specified units can involve the passing of arguments.

In a conditional non-iterative -do- the unit names "x" and "q" mean "don't do anything" and "goto q" respectively. In a conditional iterative -do- "x" means "don't do anything on this iteration", and "q" means "quit doing this statement and go on to the next statement". In other words, "x" means "fall through to the next iteration", while "q" means "fall through to the next TUTOR statement". For example,

```
        do      i-2,ua,x,q,ud,i⇐1,4
        show    i
```

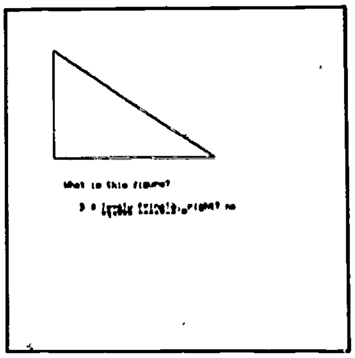
will display the number "3". For i equal to 1 we do "ua"; for i equal to 2 we do nothing; for i equal to 3 we quit and go on to the following -show- statement.

VII. Judging student responses

You now know quite a bit about how to express in the TUTOR language your instructions to PLATO on how to administer a lesson to a student. You may not have realized it, but in the process you have learned a great deal about the fundamental concepts of computer programming. You can calculate, produce complex displays, and construct rich branching structures. You have studied aspects of initialization problems, seen the importance of subroutines, and looked at some stylistic aspects of good programming practice such as defining variables, placing unit pointer commands at the head of main units, etc. With this solid background you are now ready to see in detail how to accept and judge student responses.

In Chapter I you saw a common type of judging situation in which you simply list the anticipated responses after an -arrow- statement, together with the display or other actions to be performed corresponding to the particular response. Let us see how TUTOR actually processes these judging commands. We will consider a slightly different version of the "geometry" unit:

```
unit    geometry
draw    51Ø;151Ø;154Ø;51Ø
arrow   2Ø15
at      1812
write   What is this figure?
answer  <it,is,a> (right,rt) triangle
write   Exactly right!
wrong   <it,is,a> square
write   Count the sides!
```



The order of the initial statements has been changed slightly. TUTOR starts executing this main unit by drawing the triangle. TUTOR next encounters the -arrow- command, places an arrowhead at position 2Ø15, and <u>notes where this -arrow- command is</u> (the second command in unit "geometry"). TUTOR then executes the -at- and -write- to display the text: "What is this figure?"

Finally, TUTOR reaches the -answer- command. This "judging" command is <u>useless</u> at this time because the student <u>has not entered a response</u>! There is nothing more that can be done but wait for the student to type a response and enter it by pressing NEXT. We call commands which operate on the student's response "judging" commands (such as -answer- and -wrong-). Other commands, such as -draw-, -at-, -write-, and -calc-, are called "regular" commands. We see that TUTOR must stop executing regular commands when a judging command is encountered. (This assumes the presence of an -arrow- command: an -answer- or other judging command without a preceding -arrow- is meaningless.)

When the student presses NEXT to enter his response, TUTOR looks at its notes and finds that the -arrow- was the second command in unit "geometry". TUTOR starts just beyond there looking for judging commands to process the student's response. It <u>skips</u> the regular commands -at- and -write-; these are not judging commands and are of no use at this point. It encounters the

-answer- command and compares the student response with the specifications given in the tag of the -answer- command.

If there is not an adequate match, TUTOR goes to the next command looking for a judging command that might yield a match. In this case the following command is a regular command (-write-) which is skipped. Next there is a -wrong- judging command, and if there is no match to the student's response, TUTOR keeps judging. Then comes a -write- regular command which is skipped.

Finally we come to the end of the unit without finding a matching judging command and must give a "no" judgment to this response (and possibly mark up the response with underlining and X's if the response is fairly close to that specified by the -answer- command). The process of starting from just after the -arrow- in the "judging state" will be repeated each time the student tries again with a revised response.

If, on the other hand, the response adequately matches the -answer- statement, TUTOR has found a match and can terminate the execution of judging commands. It switches to processing regular commands with the result that the following "write   Exactly right!" will be executed. (This regular command is skipped unless a match to the -answer- flips TUTOR out of the "judging state" into the "regular state".) Then TUTOR, <u>in the regular state</u>, comes to a judging command (-wrong-) which terminates the processing. TUTOR finishes up by placing an "ok" beside the student response. (Similarly, a match to the -wrong- would flip TUTOR to the regular state to execute the regular statement "write   Count the sides!")

When the -arrow- is finally "satisfied" by an "ok" judgment, TUTOR returns one last time to the -arrow- and searches for any other -arrow- commands in the unit. In this search it skips both regular and judging commands. In our particular example no other -arrow- is found, so all arrows (one!) in the unit have been satisfied. After the student has read our comment to him he presses NEXT and proceeds to the next main unit.

It may seem wasteful to you that TUTOR keeps going back to the -arrow- only to skip over the regular commands preceding the first judging command. It turns out that skipping a command is an extremely fast procedure and that keeping a single marker (the location of the -arrow- command within the unit) greatly simplifies the TUTOR machinery.

In the example, the replies "Exactly right!" or "Count the sides!" would be displayed at the location 2317, three lines below the response on the screen. This standard positioning can, of course, be altered by an -at- statement.

Here is another illustrative example. You can watch a step-by-step animation of the processing of the following unit by studying the "simulation" topic available through lesson "aids" on a PLATO terminal. The unit is:

```
unit    canine
at      21Ø5
write   Name a canine:
arrow   23Ø8
answer  dog
write   A house pet.
answer  wolf
write   A wild one!
wrong   cat
write   A feline!
```

Suppose the student enters "wolf" as his response. TUTOR initiates the "judging state" just after the -arrow-. The first -answer- (dog) does not match, so TUTOR stays in the judging state and skips the "write A house pet." There is a match to the following "answer wolf", so judging terminates and the regular state starts. The "write A wild one!" is executed, not skipped. Next TUTOR encounters a "wrong cat", and since -wrong- is a judging command, this terminates the regular state. The student gets an "ok" judgment. The search for another -arrow- does not find one, so the student has successfully completed the unit. Study the simulation in lesson "aids".

This method of processing judging and regular commands yields a readable programming structure, with judging commands delimiting the regular commands used to respond to the student. We have spent time discussing the details in order to simplify our later descriptions of the various types of judging commands used to match, modify, or store student responses.

It is important to point out in connection with these matters that the -do- and -goto- commands are <u>regular</u> commands. They are, therefore, skipped over during the judging state and during the search state (looking for a possible additional -arrow- after an arrow has been satisfied.). There is another command, -join-, which works like -do- except that the -join- command is universally executed whether TUTOR is in the regular state, the judging state, or the search state. It is, therefore, possible to -join- units containing judging commands or even containing additional -arrow- commands, whereas a -goto- or -do- is incapable of accessing other units in the judging or search states since these regular commands are skipped. Although the -do- command acts essentially like a -join-, it is, nevertheless, a regular command and is skipped during the judging and search states. Only the -join- command itself has the unique characteristic of being performed in all states--regular, judging, and search.

It is frequently useful to handle more than one response in a unit. Let's ask "Who owned Mount Vernon?", and after receiving a correct response ask in what state it is located but stay on the same page:

```
unit    wash
at      812
write   Who lived at Mount Vernon?
arrow   1015
answer  <George,G> Washington
at      1120
write   Great!
wrong   Jefferson
at      1112
write   No, he lived at Monticello.
arrow   1715
at      1512
write   In what state is it located?
answer  (Va,Virginia)
```

If you say "Jefferson" the -wrong- is matched. Regular commands are executed until you run into the second -arrow-, which ends the range of the first -arrow-. In other words, when you are working on one -arrow- the next -arrow- is a terminating marker. If you say "Washington", the student gets the "Great!" comment. Since the -arrow- is now satisfied, TUTOR starts at the first -arrow- searching for another -arrow-. In this search state all commands other than -join- are skipped (-join- may be used to attach a unit that contains another -arrow-). A second -arrow- is encountered, which changes the search state into the regular state. The arrowhead is displayed on the screen and the location of this -arrow- within the unit is noted. The regular commands following this second -arrow- are processed to display the second question. The final -answer- command stops this processing to await the student's response.

There is another way to do this which is probably more readable:

```
unit    wash
next    wash
at      812
write   Who lived at Mount Vernon?
arrow   1015
answer  <George,G> Washington
at      1120
write   Great!
wrong   Jefferson
at      1112
write   No, he lived at Monticello.
endarrow
at      1512
write   In what state is it located?
arrow   1715
answer  (Va,Virginia)
```

The -endarrow- command defines the end of commands associated with the first -arrow-. Note that -endarrow- changes the search state to regular state. One benefit of this form is that the second arrowhead appears on the screen after the text of the second question, which often seems more natural.

It may seem rather abrupt that the "Great!" and "In what state is it located?" both appear on the screen at the same time. It might be better to let the student digest the reply before presenting the second question. We might insert a -pause- with no tag just after the -endarrow-. Now TUTOR waits for you to press a key, to signal you want to go on, before presenting the next question.

The -endarrow- command is quite useful even in units which contain only one -arrow-:

```
         .
         .
         .
         arrow    1213
         answer   dog
         write    Bowwow!
         answer   wolf
         write    Howl!
         wrong    cat
         write    Meow.
         endarrow
         calc     y⇐37+y
         circle   1ØØ,25Ø,25Ø
```

The commands following the -endarrow- will be executed only after the -arrow- is satisfied, whether by the response "dog" or "wolf". So this is a convenient way to finish up the unit.

While it is possible to -join- or even -do- units which contain -arrow- commands, two seemingly arbitrary rules must be followed or you will get unpredictable results:

1) A unit attached by -join- or -do- which contains one or more -arrow- commands must end with an -endarrow- command (possibly followed by regular commands).
2) This attached unit must not contain any -goto- commands.

If you violate either of these rules, strange things will happen because TUTOR may "undo" from this unit several times (during judging, while processing regular commands, or in search state). If you follow these two rules the -join- or -do- will act like a text-insertion device: your program will act as though you had inserted the attached unit where the -join- or -do- was. We will discuss these rules in more detail later in this chapter.

Student specification of numerical parameters

The -answer- and -wrong- commands make it easy to specify a list of anticipated responses each of which, due to the specification of synonymous and optional words, can allow the student considerable latitude in the way he phrases his response. In some cases there can be no list of anticipated responses and a different technique must be used. For example, you might ask the student to specify a rocket's launch velocity and use his number to calculate and display the rocket's orbit. Or you might ask him for his name for later use in personalized messages such as "Bill, you should look at Chapter 5." In such cases all you can anticipate is that the response will be a number or a name, but you can't possibly list all possible numbers or names!

Here is an example of such a situation. We will provide the student with a desk calculator accessible on the DATA key. In the desk calculator mode he can type complicated expressions (such as "2+6^3") and receive the evaluated result.

```
unit    mainline
data    desk
at      3020
write   Press DATA for calculator
   .
   .
   .
unit    desk
next    desk          $$ for repeated use
at      1713
write   Type an expression.
        Press BACK when finished.
arrow   1915
store   eval          $$ Be sure to define "eval".
ok                    $$ Accept all responses.
write   The result is <s,eval> .
```

The -store- command will evaluate the student's expression (e.g. "13sin30°") and store the result in "eval" (in this case, the number 6.5). The -store- command is a judging command because it operates on the student's response and can be executed only after the student initiates judging by pressing NEXT. The -ok- command is a universal -answer- which matches all responses: it unconditionally flips TUTOR from the judging state to the regular state. In this example it accepts any response and enables the following -write- to display the evaluated result.

Note that a student need not use parentheses with functions: sqrt25, cos60°, arctan3 are all legal. Such expressions are illegal in a -calc-. In a moment we'll see another way in which TUTOR is more tolerant of students than of authors.

What if the response cannot be evaluated, such as "$(-3)^{1/2}$" or "19/" or "(3+5)))"? The student will get a "no" judgment. To see how this works, let's insert a -write- statement after the -store-:

```
        .
        .
        .
        store   eval
        write   Cannot evaluate!
        ok
        .
        .
        .
```

Notice that this new -write- is normally skipped because the -store- leaves us in the judging state. But if the student's expression cannot be evaluated, -store- makes a "no" judgment and switches us from the judging state to the regular state. Then TUTOR executes the "write Cannot evaluate!", after which it encounters a judging command (-ok-) which stops the regular processing. Note that -store- terminates judging only on an error condition, whereas -answer- terminates judging only on a match, and -ok- always terminates judging.

You can tell the student precisely what is wrong with his expression by use of the system variable "formok". This variable is -1 if the student's expression can be evaluated but takes one of several positive integral values for specific errors such as unbalanced parentheses, bad form unrecognized variable name, etc. The variable "formok" is defined automatically to perform this function. (If you yourself define "formok=v3" you override the system definition and you won't get these features.) The particular values assumed by "formok" can be obtained through on-line documentation at a PLATO terminal.

You can give the student some storage variables. Let's define a couple of variables for the student:

```
        define  student     $$ special define set
                bob=v3Ø,cat=v31
```

Place these defines ahead of everything else in the lesson. Suppose you have assigned bob⇐18 and cat⇐3. If the student types "2bob" he gets 36. Or he can type "bobcat" and get 54, whereas bobcat would be illegal in a -calc-, where you would need bob×cat or bob(cat). Only names defined in the set of definitions labeled "student" may be used by the student in this way. Attempted use of names in your other sets of defines will give a value of "formok" corresponding to "Unrecognized variable name".

We have discussed a desk calculator, but clearly the store/ok combination will work in any situation where we let the student choose a number. Another good example is in an index of chapter numbers:

```
unit    table
base
term    index
at      1218
write   Choose a chapter:
          1)  Introduction
          2)  Nouns
          3)  Pronouns
          4)  Verbs
arrow   1822
long    1               $$ get one digit; don't wait for NEXT
store   chapter
no
jump    chapter,x,x,intro,unoun,pron,verb,x
write   Pick a number between 1 and 4.
```

The -long- command following an -arrow- (but preceding any judging commands) sets a limit on the length of the student's response. The "long 1" is particularly useful here because the student need not press NEXT but has only to press the single number and judging starts right away. (For -long- of greater than 1 there must be an accompanying "force long" statement or else a NEXT key is required.) The reason -long- must precede any judging command is that the long specification is needed before the student starts typing, whereas we proceed past the judging command only after the student enters a response. You might think of -long- as a kind of modifier of the -arrow- command. The -arrow- sets a default maximum response length which is overridden by the following -long- statement.

The -no- in this index unit is similar to -ok- in that it unconditionally terminates judging, but it makes a "no" judgment. If "chapter" is a number from 1 to 4, the -jump- will take the student to his chosen chapter; since -jump- erases the screen the "no" will not be seen. If, however, "chapter" is not in range, we fall through the -jump- to an error message. There will be a "no" next to the response and the student must try again.

Student specification of non-numerical parameters

Having seen how to let the student specify a number, let's see how to ask him to tell us his name or nickname to permit us to speak to him by name:

```
unit    meet
at      1215
write   Hello, my name is Sam Connor.
        What's your name?
arrow   1620
long    8               $$ limit to 8 characters
storea  name            $$ define "name" earlier
ok
write   Pleased to meet you, ⊲a,name⊳ !
```

-storea- is a judging command which will store alphabetic information as distinguished from numeric information. The ⊲a,name⊳ is the embedded form of the statement "showa name" which will display alphabetic information. This unit will feed back to you any name you give it. Notice that you can't enter a name of more than 8 characters because of the -long- command. If capitalized, the name must be shorter because a capital letter counts as two characters: TUTOR stores a capital letter as a "shift" character plus the lower-case letter. (Insert a "force long" statement anywhere before the -storea- if you would like judging to start upon hitting the -long- limit, without having to press NEXT.)

A statement of the form "storea name,3" will store just the first three characters of the student's response. You can get and keep a character count of the length of the student's name, including "shift" characters, by referring to the system variable "jcount", which is a count of the number of characters in the copy of the student response used for judging--hence the "j". With these facts in mind, change the -storea- to

```
        storea  name,(namlng⇐jcount)
```

which will store the whole response and save the length. Be sure to define both "name" and "namlng", but do not define "jcount" or you will override TUTOR's definition of its function. Also, change the embedded -showa- to

```
              ⊲a,name,namlng⊳
```

to show precisely the correct number of characters.

The reason for saving the present value of "jcount" in "namlng" is that "jcount" will change at each -arrow- in the lesson, whereas throughout the lesson you will repeatedly use "showa name,namlng" or ⊲a,name,namlng⊳ to call the student by name. So you want "namlng" to keep the name length. Incidentally, a -showa- with only a single argument (such as "showa name") will show ten characters, which is the number of characters (including shift characters) that will fit in one of your variables.

It is possible to store alphabetic information longer than ten characters. Change the "long 8" to "long 20". Suppose you've defined "name=v24": you must make sure that you are not using v25, and change your defines if necessary. The 20-character name will need both v24 and v25 since each variable can hold only ten characters. With these changes it is possible to enter a long name (e.g., Benjamin Franklin, which is 19 characters counting shift characters).

Difference between numeric and alphabetic information

When we were studying the desk calculator unit, we defined for the student a variable "bob=v30". Suppose the student enters as his response the word "bob". If we use a numeric -store-, we will get the number presently contained in v3∅, which might be 529.3. If we use an alphabetic -storea-, we will just get the string of characters "bob" which is simply a name and nothing more. Perhaps the distinction is most easily seen with an example:

```
        define  student
                bob=v1
        define  ours,student      $$ include "student" set of defines
                name=v2,num=v3
        unit    test
        next    test
        calc    bob⇐π             $$ π means 3.14159........
        arrow   1815
        store   num
        storea  name
        ok
        write   num = ◁s,num▷
                name = ◁a,name,jcount▷
```

Consider various responses. "2bob" should give a numeric 2π (6.2832) and an alphabetic "2bob". More properly, we speak of "alphanumeric" information (letters and numbers) in the latter case. The response "3-4/5" yields a numeric 2.2 and an alphanumeric "3-4/5".

In other words, a storea/showa combination just feeds back exactly the alphanumeric text entered by the student. On the other hand, a -store- involves a numerical evaluation of the student's response, and a later -show- converts this numerical result into appropriate characters to display on the screen so that we can read the result. You might interchange the "num" and "name" arguments on the -store- and -storea- commands to see what weird things happen if you pair -store- with -showa- (instead of -show-) or if you pair -storea- with -show- (instead of -showa-).

To sum up, if you accept numeric information with a -store-, display it with a -show-. If you accept alphanumeric information with a -storea-, display it with a -showa-.

More on -answer- and -wrong- (includes -list- and -specs-)

There are some additional features of -answer- (and -wrong-) which should be pointed out. First, -answer- will not only handle word or sentence responses but will also handle numbers:

```
        answer  7 women <and> 5 men
```

This -answer- will be matched by a student response of the form "14/2 women and 3+2 men" because simple expressions such as 14/2 or 3+2 are evaluated by the -answer- command. Currently the -answer- command will not handle more complicated numerical expression. (Later we will discuss the -ansv- and -wrongv- commands which handle expressions as complicated as those handled by -store- but without the sentence capabilities of -answer- and -wrong-. There are also -ansu- and -wrongu- commands which are similar to -ansv- and -wrongv- but treat scientific units on a dimensional basis.)

If the student says "37 women and 5 men", the incorrect number 37 will have XXX under it, whereas the response "6.5 women and 5 men" will have the 6.5 underlined since it is nearly correct (similar to a misspelling of a word). Normally -answer- and -wrong- consider numbers off by less than 10% to be "misspelled". You can alter these specifications by preceding the list of -answer- and -wrong- commands with a -specs- command:

```
unit    trial
arrow   1815
specs   toler,nodiff
answer  7 women <and> 5 men
```

-specs- is a judging command--it affects the operation of other judging commands which follow it. Here it has been used to specify that a "tolerance" of 1% is permitted and that "no difference will be allowed for underlining" (normally 10%). Having specified both "toler" and "nodiff", any expressions within 1% of 7 and 5 will be accepted, but expressions with larger discrepancies will not be underlined.

Note carefully that since -specs- is a judging command, it terminates the processing of regular commands. Among other things this means that a -long- command must precede the -specs-, not follow it. If -long- comes after -specs-, TUTOR won't realize it is supposed to prevent the student from entering a longer response, since it won't have seen the -long- before stopping to wait for the student's response.

Here are some other useful applications of -specs-:

```
specs   bumpshift,okspell
answer  the antidisestablishmentarianism doctrine
```

This changes student's capital letters to small letters, and specifies that misspellings are to be considered ok. Note that an -answer- tag should not contain capital letters if you use "specs bumpshift" to uncapitalize the student's capital letters.

```
specs   okextra
answer  Washington
```

This says it is ok to have extra words, so that "It was George Washington" will be an acceptable response.

```
specs   noorder
answer  apples pears and peaches
```

This specifies that no particular word order is required. Note the absence of commas in the -answer- tag: currently such punctuation marks are not allowed there, but all punctuation marks are ignored in the student's response, so he may use commas. There exists a much less powerful -exact- command as well as other techniques for judging particular punctuation when that is necessary.

```
specs   nookno
ok
```

Here we specify that no "ok" or "no" be displayed beside the student's response, contrary to the normal situation.

For other -specs- capabilities see reference material described in Appendix A.

There is another important feature of -specs- in addition to its use in specifying various options: it marks a place to return to after judging. Consider the following unit. You do not define the system variable "spell".

```
unit    presi
at      1212
write   Name one of the first three U.S. presidents.
arrow   1510
specs   bumpshift
at      2508
writec  spell,No misspellings!,
        Underlining indicates a misspelled word.
answer  washington
write   Good old George.
answer  adams
answer  jefferson
```

Suppose the student types "WASHINGTON". TUTOR starts judging just after the -arrow- and encounters -specs-, a judging command. The tag tells TUTOR to change the response to "washington" for judging purposes. (Incidentally, this operation changes "jcount", the character count of the judging copy of the student's response, from 20 to 10 because the "shift" characters are knocked out.) Moreover, TUTOR makes a note that it encountered a -specs- command as the fourth command in unit "presi", and this marker will be used in a moment. TUTOR skips the following -at- and -writec- because regular commands are skipped in the judging state.

Next, TUTOR encounters "answer washington" which matches the student's (altered) response, and this terminates judging. The succeeding regular commands are processed as usual. In this case there is only a "write Good old George" before we run into another judging command ("answer adams") which stops the processing.

Well, not quite stops processing. For simplicity we've lied a little up till now! At this point TUTOR asks one last question: "Did I pass a -specs- command in processing this response?" The answer is yes--at the fourth command in unit "presi". TUTOR now processes any regular commands following that -specs- marker. In this case TUTOR does an "at 25Ø8" and a -writec- before finally being stopped (really stopped this time) by the first -answer- command.

The -writec- refers to the system variable "spell" which is true (-1) if spelling is fine, false(Ø) if a misspelling has been detected. "spell" is -1 if there are no underlined words, but there may be X'ed words--words that are completely different.

The usefulness of the marker property of -specs- is that you can specify a central place to put messages and calculations which should be done no matter which judging command is matched. We will see additional applications of this useful feature of the -specs-. Notice that a later -specs- command will override an earlier -specs- marker in a manner analogous to the way a later -help- command overrides an earlier setting of the "help" marker. Note too that if no regular commands follow the -specs-, TUTOR finds nothing to do when it comes there after being nearly stopped as described above. This was the situation in our previous examples such as

```
        specs    nookno
        ok
```

where there are no regular commands between the -specs- and the -ok-.

Let us return for a moment to the -answer- command. We had examples involving synonyms such as (right,rt) or (Va,Virginia). A convenient way to specify synonym lists which occur frequently in a lesson is to define a -list-:

```
        list    affirm,yes,ok,yep,yeah,sure,certainly
```

Here "affirm" is the title of a list of synonyms; "affirm" is not itself a member of that list. With this definition, which should be placed at the very beginning of your lesson along with your -define- statement, you can write

```
        answer  ((affirm))
        wrong   maybe ((affirm))
```

These are equivalent to

```
        answer  (yes,ok,yep,yeah,sure,certainly)
        wrong   maybe (yes,ok,yep,yeah,sure,certainly)
```

Note that "answer we affirm" does not imply this list of synonyms, as a single important word by itself does not refer to a list. You can use the list equally well to specify optional words, as in

```
        answer  <<affirm>> it is
```

<<affirm>> is equivalent to <yes,ok,yep,yeah,sure,certainly>. Note that <affirm> merely refers to the single word "affirm". Double marks are needed to refer to the list whose title is "affirm". You can combine references to synonym lists with individual words:

```
        wrong   usually (definite, (affirm))
        answer  often <definite, <affirm>>
```

Another list which might be particularly useful is this one:

```
        list    negate,no,nope,not,never,huhuh
```

This covers the main capabilities of the -answer- and -wrong- commands and their associated -list- definitions. The -specs- command may be used to modify how -answer- works and also serves as a useful marker. The marker function of -specs- is not limited to -answer- but holds for any judging commands which follow it, including -ok- and -no-.

The -answer- (or -wrong-) command can nicely handle responses which involve a relatively small vocabulary of words. It is, therefore, adequate when the context limits the diversity of student responses, such as foreign language translation drills where there are only a few permissible translations of the sentence and each such sentence contains a rather small number of allowable words. The detailed markup of the response is a particularly useful feedback to the student in such a drill.

The -answer- command is not well suited to a more free dialog with the student where the context is broader and where the vocabulary used by the student may encompass hundreds of words. In the next section we discuss the -concept- command which can cope with more complexity.

Building dialogs with -concept- and -vocabs-

An excellent example of a dialog is a lesson on qualitative organic chemistry analysis written by Prof. Stanley Smith of the Department of Chemistry, University of Illinois--Urbana. This lesson helps students practice their deductive skills on PLATO before performing in a laboratory the identification of an unknown compound. Prof. Smith has PLATO choose at random one of several organic compounds and then invites the student to ask experimentally-oriented questions aimed at identifying the unknown. Typical questions are "what is the melting point"; "does it dissolve in sulfuric acid"; "show me the infrared spectrum"; "is it soluble in H_2O". There are over a hundred such concepts important in this simulated laboratory situation, and since each concept has many equivalent forms drawing upon a vocabulary of hundreds of words, the number of possible responses is astronomical. How can this be handled?

Although the context is far broader than that of a language drill, it is, nevertheless, sufficiently limited to be tractable: No attempt is made to recognize arbitrary student responses such as "cook me some apple pie". With this quite reasonable restriction the situation can be handled by using the -vocabs- command (analogous to -list-) to define a large vocabulary, with appropriate "synonymization", associated with a list of -concept- commands (analogous to -answer-) which express the basic concepts meaningful in the context of this lesson.

Here is a fragment of the -vocabs- command:

```
vocabs  labtest     $$ vocabulary must have a name
        <is,it,a,does,in,what>
        (color,red,blue,green)
        (water,H2O)
        (dissolve,soluble)
          .
          .
          .
```

And here are a couple of the many -concept- commands:

```
:
arrow   1213
concept what color
write   It is red.
concept soluble in water
write   It's slightly soluble in water.
:
:
:
```

Consider what TUTOR does with "concept soluble in water". It knows that -concept- has a tag consisting of words defined by a previous -vocabs-. (As usual with such matters, the -vocabs- should be at the beginning of the lesson.) The first word in the tag is "soluble" which it finds is the third important word in the vocabulary, discounting the ignorable or optional words "is,it,a" etc., and lumping synonyms together so that "dissolve" is also considered a "number 3" vocabulary word. The next word of the tag is "in" which TUTOR throws away because the -vocabs- says that word is ignorable.

The next word is "water", which is in the second set of important -vocabs-synonyms. The net result is that "concept soluble in water" is converted to the sequence "3 2".

Now consider a student in this lesson who types "does it dissolve in H_2O". Superfically this looks quite different from the -concept- tag "soluble in water". But TUTOR encounters a -concept- command which, unlike -answer-, indicates that the student's response should be looked up in the defined vocabulary (in the case of -answer- there is no one vocabulary set because each -answer- may include various -list- references and particular words specific to that -answer-.) By a process identical to the conversion of the author's -concept- tag, TUTOR converts the student's response into "3 2". This compact form "3 2" does not match the first "concept what color" (which was converted to "1") so TUTOR proceeds to the next judging command, which is "concept soluble in water" or rather its converted form "3 2". This matches, so judging terminates and regular processing begins. The student gets a reply "It's slightly soluble in water."

Notice that the first -concept- encountered triggers the transformation of the student's response into the compact form suitable for looking through a very long list of concepts. If the -vocabs- contains an entry such as (five,5,cinco), the student may match this entry with "3+2", just as in an -answer- statement involving numbers.

You will have to experiment a little with this machinery to learn how best to manage the synonymization in the vocabulary. This does depend on the context. In an art lesson it would be disastrous to call red and blue synonyms as was done here, but it makes sense in this context where the only concept related to color has to do with "what color is it", which means essentially the same as "is it red" or "is it blue".

You will find that the use of words not defined by -vocabs- will result in a markup indicating which words are undefined (you will see uuuuu for unknown under these words). If your context is such that you need worry only about key words and don't care if the student asks "does it dissolve superbly in water", you might precede the first -concept- with a "specs okxvocab" which says that extra student words not found in the vocabulary may be ignored, as though they had been so specified in the -vocabs- tag. In that case you need not define any ignorable words with -vocabs-, but you would write "concept dissolve water", not "concept dissolve in water" since extra author words are not tolerated. If you don't use "specs okxvocab", the student's word "superbly" will be marked (uuuuuuuu). If the student misspells a vocabulary word, that word will be underlined: saluble in water.

Here is an alternative and more detailed version of the heart of the dialog lesson, which illustrates several points. This is a rather complex example which brings together many aspects of TUTOR. Note particularly that the -concept- statements now are listed one after another. The variable "unknown" is a number from 1 to 4 associated with which compound the student is attempting to identify. The system variable "anscnt" is set to zero when judging starts (and when a -specs- is encountered) and it counts the number of -answer-, -wrong-, -ok-, -no-, and -concept- commands passed through. If the third such command terminates judging, "anscnt" will have the value 3. If no match is found, "anscnt" is set to -1.

```
        :
        :
        arrow    1213
        wrong    what is it
        write    That is for you to determine!
        specs             $$ to clear anscnt again
        goto     anscnt>Ø,unknown,x
        writec   vocab,I don't understand your sentence.,
                 The uuuu words are not in my vocabulary.
        concept  what color
        concept  soluble in water
        concept  boiling point
        :
        :
        unit     unknown
        goto     unknown-2,reply1,reply2,reply3,reply4
        *
        unit     reply1
        writec   anscnt,,,It is colorless.,
                 It is slightly soluble in water.,
                 The boiling point is 245-247° C.,
        :
        :
```

The statement "wrong what is it" is necessary because a "concept what is it" contains only ignorable words and would, therefore, not distinguish between "what is it" and "does it what", which also contain only ignorable words. Since -specs- resets "anscnt" to zero, "anscnt" will have the value 2 if the student's response matches the second -concept- ("soluble in water"). No regular commands follow this -concept-, so TUTOR goes right to the -specs- marker to execute the regular commands there. Since "anscnt" is greater than zero, TUTOR does a -goto- to unit "unknown", where there is a -goto- to unit "reply1" (assuming we are working on unknown number 1), which writes "It is slightly soluble in water" on the student's screen.

This structure makes it very easy to add a fifth unknown compound to the lesson. The -vocabs- and list of -concept- commands do not have to be changed, since the basic concepts and vocabulary are pertinent to the analysis of any compound. All that is necessary is to add "reply5" to the end of the conditional -goto- in unit "unknown" and to write a unit "reply5" patterned after unit "reply1". All done!

What happens if the student says "it what does"? This will not match the -wrong- nor any of the -concept- commands, so "anscnt" will be -1. Therefore, the -goto- just after the -specs- will fall through to the following -writec-, which gives one of the two messages dependent on the system variable "vocab" (true if all words found in vocabulary, false if some words not found--those having uuuu underneath them.) In this case the student will get the message "I don't understand your sentence", whereas if he says "what is elephant" he will see the uuuu's under "elephant" and get the message "The uuuu words are not in my vocabulary".

That was a fairly complicated example, but the discussion is justified by the general usefulness of many of the techniques employed and by the extraordinary power such a structure yields, both in its sophisticated handling of student responses and in the ease of expansion to additional options.

Suppose the -arrow- is in a unit "analysis". One way to proceed from one question to the next would be to place a "next analysis" in this unit. There is an elegant way to avoid erasing and recreating the display associated with this unit. Instead of proceeding, let's judge each response "wrong" so that we stay at this -arrow-. Replace the -specs- command with these two statements:

```
      :
      :
      :
      specs    nookno      $$ so "no" doesn't appear
      judge    wrong
      :
      :
```

Despite its name, -judge- is a regular command, not a judging command. It can be used to alter the judgment made by the judging commands. In this case, TUTOR first skips over this regular command to get to the -concept- commands. If one of these matches the student response, TUTOR makes an "ok" judgment, but upon going to the -specs- marker TUTOR finds a "judge wrong" which overrides the earlier judgment. TUTOR keeps going, processing regular commands, and produces a message as we have seen before. The "nookno" specification prevents a "no" from appearing on the screen and the student simply sees our message to him. But the -arrow- has not been satisfied, so when he presses NEXT, TUTOR erases the response and awaits a new response. Each time, the student gets a reply to his experimental question, and the "wrong" judgment takes us back to the -arrow-.

This is a good way to manage the screen because only a small portion of the display changes: the surrounding text and figures remain untouched. The "next analysis" re-entry to this same main unit would quickly get tiresome because of the repetitious replotting of the surrounding material.

You now should be able to use -answer-, -wrong-, and -list- in situations where the vocabulary is small and -concept- and -vocabs- where the vocabulary is large. You have seen how to use -specs- both to specify various judging options and to mark a place where post-judging actions can be centralized. You have seen one form of the regular -judge- command: "judge wrong". overrides an "ok" judgment made by an -answer- or -concept-. In fact, while -wrong- is the appropriate opposite of -answer-, currently the "opposite" of -concept- must be implemented by a -concept- followed by a "judge, wrong" since there is no "wrong" form of the -concept- command.

There is a particularly convenient way to make different concepts equivalent, including different word orders:

```
        concept dissolve in water
                water soluble
                drop in water
        write   It's soluble in H2O.
```

The "continued" -concept- specifies synonymous concepts. If the student's response matches any of these three concepts the same message will be given. Also, "anscnt" will be the same no matter which of these makes the match.

Use of -vocabs- makes possible the underlining of misspelled vocabulary words (or their acceptance with a "specs okspell"), just as with the -answer- command. Similarly, "specs noorder" can be used to specify that no particular word order is required. There is a -vocab- command which permits a larger vocabulary at the price of giving up these spelling and order capabilities. There is an -endings- command which makes it easy to expand a vocabulary in terms of word roots plus endings.

The -judge- command

We have encountered the regular command -judge- (not a judging command!) and seen one use of it to "judge wrong" a response that had already received an "ok" judgment. The -judge- command may also be used to "judge ok", a response independent of what a previous judging command may have had to say. There is a conditional form:

```
        judge   3x-y,ok,x,wrong
```

will make the judgment "ok", or not alter the current judgment (the "x" option), or make the judgment "wrong" depending on the condition "3x-y".

Here is a useful example:

```
unit     negative
at       1214
write    Give me a
         negative number:
arrow    1516
store    num
write    Cannot evaluate your expression.
ok                      $$ terminate judging
judge    num<Ø,ok,wrong
writec   num<Ø,Good!,That's positive!
```

We could just as well write "judge num<Ø,x,wrong" since the original judgment was a universal 'ok'. (Later we will study -ansv- and -wrongv- which are also useful in numerical judging.)

We have been using the -ok- or -no- commands to terminate judging unconditionally, as in the last example. It is sometimes useful to be able to switch in the other direction, from the regular state to the judging state. For example, suppose you want to count the number of attempts the student makes to satisfy the -arrow-:

```
:
:
calc     attempt⇐Ø
arrow    1518
ok
calc     attempt⇐attempt+1
judge    continue
answer   cat
   etc.
```

Judging starts just after the -arrow-. The -ok- terminates judging to permit executing the regular -calc- which increments the "attempt" counter. Then the regular -judge- command says "continue judging", which switches TUTOR back into the judging state to examine the -answer- and other judging commands which follow. If the response is finally judged "no", the student will respond again, and since judging starts each time from the -arrow-, the "attempt" counter will record each try.

A COMMON MISTAKE is to leave out the -ok- and "judge continue" which permit counting each attempt. If you write

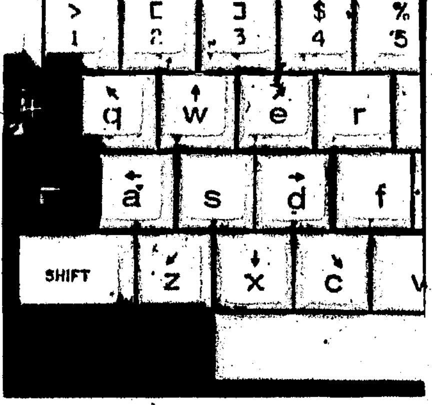
```
:
:
:
calc     attempt⇐Ø
arrow    1518
calc     attempt⇐attempt+1
answer   cat
:
:
```

then "attempt" will stop at one. TUTOR initializes "attempt" to Ø, then encounters the -arrow- and notes its position in the unit. Then the following -calc- increments "attempt" to 1, after which the -answer- judging command terminates this regular processing to await the student's response. The student then enters his response and TUTOR starts judging. The first command after the -arrow- is the incrementing -calc-, which is skipped because it is a <u>regular</u> command and TUTOR is looking for judging commands. This will happen on each response entry, so "attempt" never gets larger than one. This explains the importance of bracketing the -calc- with -ok- and "judge continue".

A related option is "judge rejudge" which is similar to "judge continue". We have seen that "specs bumpshift" alters the "judging copy" of the response by knocking out the shift characters. The judging copy is the version of the response which is examined by the judging commands such as -answer-. This version may differ from the student's actual response due to various operations such as "specs bumpshift". It is also possible to -bump- other characters or to -put- one string of characters in place of another. All such operations affect the judging copy <u>only</u> and do not touch the original response, which remains unmodified. The statement "judge rejudge" replaces the judging copy of the response with the original response, thus cancelling the effects of any previous modifications of the judging copy. It also initializes the system variables associated with judging, including "anscnt". It is, therefore, much more drastic than "judge continue", which merely switches TUTOR to the judging state without affecting the judging copy or the system variables.

Another exceedingly useful -judge- option is "judge ignore" which erases the student's response from the screen and permits him to type another response without first having to use NEXT or ERASE. Unlike "judge wrong", "ok", or "continue", "judge ignore" stops all processing and awaits new student input. Even the commands following a -specs- won't be performed. On the other hand, TUTOR goes on to the following commands after processing -judge- with tags "ok", "wrong", or "continue".

A good example of the heightened interaction possible through the use of "judge ignore" is the following routine which permits the student to move a cursor on the screen. We use the typewriter keys d,e,w,q,a,z,x, and c, which are clustered around a 3 by 3 square on the keyboard, to indicate the eight compass directions for the cursor to move on the screen. These keys have small arrows on them to indicate their common use for moving a cursor.



```
unit     cursor
calc     x⇐y⇐25∅         $$ initialize cursor position
         dx⇐dy⇐1∅        $$ cursor step size
do       plot            $$ plot cursor on screen
inhibit  arrow           $$ don't show the arrowhead
arrow    32∅1
long     1
specs                    $$ come here after judging
goto     move            $$ -goto- is a regular command
answer   d               $$ east:   anscnt=1
answer   e               $$ northeast      2
answer   w               $$ north          3
answer   q               $$ northwest      4
answer   a               $$ west           5
answer   z               $$ southwest      6
answer   x               $$ south          7
answer   c               $$ southeast      8
ignore                   $$ equivalent to: { no
*                                          { judge   ignore
unit     move
* erase old cursor
mode     erase
do       plot
mode     write
* increment x and y on the basis of "anscnt"
calcs    anscnt-2,x⇐x+dx,x+dx,x,x-dx,x-dx,x-dx,
         x,x+dx
calcs    anscnt-2,y⇐y,y+dy,y+dy,y+dy,y,
         y-dy,y-dy,y-dy
do       plot
judge    ignore
*
unit     plot
at       x,y
write    +               $$ use "+" for cursor
```

This routine permits the student to move the cursor rapidly in any direction on the screen. A letter which matches one of the -answer- statements will cause the -calcs- statements to update x and y appropriately to move in one of the eight compass directions. The "long 1" makes it unnecessary to press NEXT to initiate judging, and the "judge ignore" after the replotting of the cursor leaves TUTOR again awaiting a new response. The "judge ignore" greatly simplifies repetitive response handling such as arises in this example. Normally such a cursor-moving routine would be associated with options to perform some action, such as drawing a line. This makes it possible for the student to draw figures on the screen.

In addition to the -judge- options discussed above, there is a "judge exit" which throws away the NEXT or timeup key that had initiated judging. This leaves the student in a state to type another letter on the end of his response. This can be used to achieve special timing and animation effects.

To summarize, the -judge- command is a regular command used for controlling various judging aspects. The -ok-, -no-, and -ignore- are judging commands which somewhat parallel the "judge ok", "judge no", and "judge ignore" options. The "judge rejudge" and "judge continue" options make it possible to switch from the regular state to the judging state with or without reinitializing the judging copy of the student response and the system variables associated with judging. All of these options may appear in a conditional -judge-:

```
        judge   expr,no,x,ok,continue,wrong,rejudge,x,ignore,ok
```

with "x" meaning "do nothing". The subtle difference between "judge wrong" and "judge no" will be discussed in the chapter on "Student Response Data". Basically, "judge wrong" is used to indicate an anticipated specific wrong response, whereas "judge no" indicates an unanticipated student response. Additional -judge- options are "quit", "okquit", and "noquit".

Finding key words: the -match- and -storen- commands

The -match- command, a judging command, makes it easy to look for key words in a student's response. Not only will it find a word in the midst of a sentence, but it replaces the found word in the judging copy with spaces to facilitate the further use of additional judging commands, including -match-, to analyze the remainder of the response. Here is the form of a -match- statement:

```
        match   num,dog,(cat,feline),horse,(pig,hog,swine)
                     Ø       1          2         3
```

Here "num" is a variable which will be set to -1 if none of the listed words appear in the student's response, to Ø if "dog" appears, to 1 if "cat" or "feline" is present, 2 if "horse" is in the response, etc. In any case, -match- terminates judging, with a "no" judgment if num=-1 or "ok" otherwise. What if more than one of the words appears in the student's response? Suppose the student says

```
        "horse and dog"
```

In this case "num" will be set to 2 because in looking at the first student word we find a match (horse). The judging copy of the response is altered by replacing "horse" with spaces so that it looks like

```
        "      and dog"
```

If we were to execute the same -match- again we would get the number Ø corresponding to "dog", and the judging copy would then look like

```
        "      and    "
```

Note that -match- always terminates judging, so that a "judge continue" is needed before another -match- can be executed. Also note that the key-words are pulled out in the order in which they appear in the student's response, not in the order they appear in the -match- statement.

Now let's do some useful things with -match-. First, we can improve greatly on our cursor program:

```
        :
        :
        inhibit arrow
        arrow    32Ø1
        long     1
        match    num,d,e,w,q,a,z,x,c
        goto     num,x,move
        judge    ignore
        :
        :
```

with unit "move" unchanged except to replace (in two places) the expression "anscnt-2" by the expression "num-1". We see that -match- is useful for converting a word to a number which represents the word's position in a list.

Another good use of -match- is in an index:

```
        unit     table
        base
        term     index
        at       1218
        write    Choose a chapter:
                   a)  Introduction
                   b)  Nouns
                   c)  Pronouns
                   d)  Verbs
        arrow    1822
        long     1
        match    chapter,a,b,c,d
        calc     chapter⇐chapter+1
        jump     chapter,x,x,intro,unoun,pron,verb,x
        write    Pick a,b,c, or d.
```

Notice that we must increment "chapter" by one if we want topic "a" to be chapter 1, since -match- associates Ø with the first element in its list (because -1 is reserved for the case where no match is found). If no match is found, there is a "no" judgment.

These applications barely scratch the surface of the power available through -match-. Here are some other ideas:

1) Use -match- to pull out negation words such as no, not, never, etc. "judge continue" and use -answer- or -concept- commands to analyze the remainder of the response. You can in this way separate the basic concept from whether it is negated, with the negation information held in the -match- variable for easy use in conditional statements.

2) -match- is useful in identifying and removing a directive before processing the rest of the information. This comes up in simulating computer compilers, in games ("move" or "capture" such-and-such), etc.

A related command is -storen-, which will find a simple numeric expression in a sentence, store it in your specified variable, and replace the expression with spaces. This is particularly useful for pulling out several numbers; -store- will handle much more complicated expressions including variables as well as numbers, but can get only one number. For example, the student might respond to a question about graph-plotting coordinates with "32.7,38.3". These two numbers can be acquired by:

```
    :
    :
    arrow   1215
    storen  x
    write   You haven't given me numbers.
    storen  y
    write   You only gave me one number.
    answer              $$ remainder should be essentially blank
    no
    write   There should just be two numbers.
```

Like -store-, -storen- will terminate judging on an error condition (that no number was found). In the example, the first -storen- removes and stores one number in "x" and the second -storen- looks for a remaining number to store in "y". The first -storen- will terminate judging if there are no numbers--the second -storen- if there is no number remaining after one has been removed. The blank -answer- will be matched if only punctuation such as commas remains after the actions of the two -storen-s.

Numerical and algebraic judging: -ansv- and -wrongv-

We have already had some experience in handling numerical and algebraic responses by using -store- to evaluate numerically the student's expression. The -ansv- (for "answer is variable") and -wrongv- judging commands evaluate the student's expression in the same way as -store- and also perform a comparison with a specified value.

The -ansv- command is useful in association with -store-. If you ask the student for a chapter number or a launch velocity of a moon rocket, it is convenient to use -ansv- to check whether his number is within the range you allow. For example:

```
    :
    :
    arrow   1314
    store   chapter
    ansv    5,4
    no
    write   Choose a chapter from 1 to 9.
    :
    :
```

Another common use is in arithmetic drills;

```
        define  b=v1,c=v2
        unit    drill       $$ multiplication drill
        next    drill
        randu   b,1Ø        $$ pick an integer from 1 to 1Ø
        randu   c,1Ø        $$ pick another integer
        at      1513
        write   What is ⊲s,b⊳ times ⊲s,c⊳ ?
        arrow   1715
        ansv    b×c         $$ no tolerance
        write   Right!
        wrongv  b+c
        write   You added.
        wrongv  b×c,1       $$ plus or minus 1
        write   You are off by 1.
        wrongv  b×c,2Ø%     $$ plus or minus 2Ø%
        write   You are fairly close.
        no
        write   You are way off!
```

The drill as written will run forever. It could be modified to stop after 5 straight correct responses, or after some other criterion has been met. Note that the response "bc" or "b×c" is judged "no" (unless you define these variables in the "student" set of defines). It is, however, humorous that the student need not do any mental multiplication, for if he is asked to multiply 7 times 9 he can respond with 7×9, which matches the -ansv-!

Let's make a change to require some multiplication on the part of the student"

```
        :
        :
        :
        ansv    b×c
        judge   opcnt=Ø,ok,wrong
        writec  opcnt=Ø,Right!,Multiply!
        wrongv  b+c
        :
        :
```

Do not define "opcnt": it is a system variable which counts the number of operations in the student's response. If the student says "7(5+8+3)/2" then "opcnt" will be 4 because there are

1) an (implied) multiplication (7 times a parenthesized expression);
2) two additions;
3) a division.

In this drill we want the student to give the result with no operations, so "opcnt" should be zero. ("specs noops,novars" can also be used to prevent the student from using operations or variables in his response.)

Recall that the first -concept- command encountered will trigger the reduction of the student's response to a compact form, through the use of the -vocabs-. This compact form can be compared rapidly against all succeeding -concept- commands. Similarly, the first -store- or -ansv- or -wrongv- causes TUTOR to "compile" the student's expression into a form which can be evaluated extremely rapidly when another of these commands is encountered. It is during the compilation process that "opcnt" is set. Just as the -vocabs- list tells TUTOR how to interpret the student's words, so the "define student" set of names tells TUTOR how to treat names encountered in the compilation of a student's algebraic response. So there are many parallels between -ansv- and "define student" on the one hand and -concept- and -vocabs- on the other.

Let's look at an algebraic example, as opposed to the numerical examples we have treated:

```
define  student
        x=v1
unit    simplify
at      1215
write   simplify the expression
                3x + 7 + 2x - 5
randu   x               $$ pick a fraction between Ø and 1
calc    x⇐x+1           $$ change to 1 to 2 range
arrow   1418
ansv    5x+2            $$ Ø tolerance
goto    varcnt-1,toofew,x,manyvar       $$ how many x's
goto    opcnt-2,toofew,x,manyop         $$ how many operations
wrongv  5x+12
write   You should subtract 5, not add it.
no
goto    formok,x,tellerr
*
unit    toofew
write   Your expression is not sufficiently general.
judge   wrong
*
unit    manyvar
write   "x" should appear only once.
judge   wrong
*
unit    manyop
write   Not simplest form.
judge   wrong
```

Unit "tellerr" would contain a -writec- involving the system variable "formok" to tell the student precisely why his expression could not be evaluated. There could be several -wrongv- statements in the example to check for various specific errors. The system variable "varcnt" during compilation of the student's expression counts the number of references to variables. For example, "x+3x+x+2" is numerically equivalent to (5x+2), so that this response will match the -ansv-, but "varcnt" will be 3 because "x" is mentioned three times. If both x and y were defined, the expression "2x+y+4x" would yield a "varcnt" of 3 (two x's and one y) and an "opcnt" of 4 (two implied multiplications and two additions).

In this way "opcnt" and "varcnt" may be used to distinguish among equivalent algebraic responses which differ only in form. Roughly speaking, what is usually called "simplest algebraic form" often corresponds to the smallest possible values of "opcnt" and "varcnt".

There are some minor technical points in the above example. -randu- with only one argument produces a fraction between Ø and 1. If this should happen to be very close to Ø then "x" would be unimportant in the expression (5x+2), so it seems better to add one to make "x" have a value between 1 and 2, which is comparable to the other quantities in the expression. We could have used the two-argument form (e.g., "randu x,8") to pick an <u>integer</u> value for "x". But suppose TUTOR chooses the integer 2 for "x": then a student who happens to give "12" as his response will match the -ansv- by accident since 5x+2 = 5×2+2 = 1Ø+2 = 12. On the other hand, with TUTOR picking a <u>fraction</u> the student would have to type something like "8.93172462173" to accidentally match the -ansv-. This just won't happen. You would have to type different numbers 24 hours a day for hundreds of years to match accidentally. If you want even more security against an accidental match, just change the value of "x" and check again. In skeleton form:

```
ansv    5x+2
goto    varcnt-1,toofew,x,manyvar
goto    opcnt-2,toofew,checkup,manyop
wrongv  5x+12
:
:
unit    checkup
randu   x               $$ new value of x
calc    x⇐x+1
judge   continue
ansv    5x+2            $$ try again
:
:
:
```

A further check is that we require exactly one "x" and exactly two operations.

There is a way to give detailed feedback to the student in case his expression is not algebraically equivalent to the desired expression (5x+2).

Suppose his incorrect expression is "6x+2", and that you have done a -storea- to save the response and a -store- to evaluate it for some integer value of x. Then ask the student this question:

```
        :
        :
        :
        write   What is the numerical value of
                 3(<s,x>)+7+2(<s,x>)-5?
```

If x is 4, this will appear on the screen as:

```
                What is the numerical value of
                  3(4)+7+2(4)-5?
```

Many students can handle a numerical example even if an algebraic example gives them trouble, so this student is likely to reply correctly, either with or without some help, that this expression gives 22. You can then reply to him with this statement (assuming his alphanumeric response is in "string" and its value in "result"):

```
        write   But your expression, <a,string,count> ,
                gives <s,result> in this case.
```

If his response was "6x+2", with a value of 26 if x is 4, this appears on the screen as

```
                But your expression, 6x+2,
                gives 26 in this case.
```

The student now sees that his expression "6x+2" does not give the value 22 which it should in the case where x is 4. You have fed back to him his own expression, evaluated for a particular case where he can see there is a conflict. In other words, anything he says may be used against him! There is here an opportunity for the student to learn by example a useful technique in simplifying complicated expressions: try some numerical cases for which you know the results and see whether they agree with the simplified expression.

It is possible to judge equations as well as expressions. Suppose we ask the student to simplify the equation "4x+3=x+12y-5". A suitable response might be "12y=3x+8" or "x=(12y-8)/3". Every time the student enters a response, let TUTOR pick a random value for the independent variable x, and calculate the corresponding value of the dependent variable y:

y⇐(3x+8)/12. Then any correct equation will be true (with value -1), and an incorrect equation will be false (with value Ø). Here is a unit embodying these concepts:

```
define   student,x=v1,y=v2
unit     equate
at       1215
write    Simplify the equation
          4x+3=x+12y-5
arrow    1718
ok
randu    x             $$ random x on each judging
calc     x⇐x+1
         y⇐(3x+8)/12    $$ y depends on x
judge    continue
ansv     -1            $$ logical true
goto     ident
wrongv   Ø             $$ logical false
write    That is false.
no                     $$ anything else
write    Give me an equation!
*
unit     ident
calc     y⇐3.72y       $$ change y arbitrarily
judge    continue
wrongv   -1            $$ should not now be true
write    That is an identity!
ok
judge    varcnt>2,wrong,ok
writec   varcnt>2,Not simplified.,Fine.
```

If the student writes "3+4", this expression has the numerical value 7, so the reply is "Give me an equation!"

If the student writes "3=4", this expression has the numerical value Ø. since it is logically false, and the reply is "That is false."

If the student writes "3^2+5=17-3", which is equivalent to 14=14, TUTOR replies "That is an identity!" The student's response is true; 14 does equal 14, so that this true relationship has the value -1, which matches the -ansv- statement. There follows a "goto ident", where the dependent variable y is changed so that it no longer bears the correct relationship to x. If the student's response had been a correct simplification of the given equation, his expression would no longer be true (-1), since y is no longer the correct function of x. In the case of "3^2+5=17-3", however, changing y has no effect and the value is still -1, which matches the -wrongv- statement in unit "ident". The student gets the message "That is an identity!"

Only if the student enters a non-identical equation will he get an "ok" judgment. Note the check on "varcnt". There could also be a check on "opcnt".

To summarize, -ansv- and -wrongv- are extremely powerful commands for algebraic or numeric responses, particularly in association with variables defined in the "define student" set. The system variables "opcnt" and "varcnt" give you additional information about the form of the response.

CAUTION: Since TUTOR performs multiplications before divisions (unless parentheses intervene), a student response of "1/2x" is taken to mean "1/(2x)", whereas the student might have in mind "(1/2)x". It is important to warn your students of this convention at the beginning of a lesson which uses algebraic judging. Scientific journals and most textbooks follow this same convention, but many students are unaware of this. Usually printed materials use the forms $\frac{x}{2}$ or $\frac{1}{2}x$ or $\frac{1}{2x}$. These forms avoid the ambiguities that arise from the slash (/) or quotient sign (÷) used on a single typewritten line. It is hoped that eventually TUTOR will make it easy for students to type fractions with the horizontal bar rather than the slash or quotient sign. Until then it is important to point out this convention to your students.

Handling scientific units: -ansu-, -wrongu-, and -storeu-

Suppose you want to ask the student for the density of mercury. A correct answer would be "13.6 grams/cm^3", but there are many equivalent ways to write the same thing. For example, the student might write "13.6×10^{-3}kg/(.01 meter)^3" or "13.6 gm-cm^{-3}", and both of these responses are equivalent to "13.6 grams/cm^3. TUTOR provides a convenient way not only to judge such responses appropriately but to give the student specific feedback if he makes specific errors such as omitting the units or giving the right units but the wrong number.

The TUTOR scheme is based on the judging performed by human instructors when grading exam questions involving numbers and units. The instructor makes two separate checks, one for the numerical value and the other for the dimensionality of the units. The dimensionality of density is (mass)^1(length)$^{-3}$, and it is the powers (1,-3) that we are interested in as well as the number 13.6. All of the equivalent correct responses listed above have a numerical value of 13.6 (in the gram-cm system of units) and a mass-length dimensionality of (1,-3). The -storeu- command (-store- with units) can be used to get the numerical part and the dimensionality if we define the units appropriately:

```
define  student        $$ units will be used by student
        units,gm,cm    $$ can define up to 1ø basic units
        gram=gm,grams=gm,kg=1øøøgm       $$ synonyms
        meter=1øøcm,cc=cm3
define  mine,student    $$ include student define set
        num=v1,dimens(n)=v(1+n)         $$ see "Arrays", chapter IX
unit    dense
at      1215
write   What is the density of mercury?
        (Include units!)
arrow   1618
storeu  num,dimens(1)
write   Cannot evaluate.
no
goto    num≠13.6,badnum,x
goto    dimens(1)≠1,badmass,x
goto    dimens(2)≠-3,badleng,x
judge   ok
write   Good!
```

We will go to a unit "badnum", "badmass", or "badleng" (not shown here) if there is something wrong with number, mass, or length. The -storeu- command has two variables in its tag: the first will get the numerical part of the student's response , and the second (dimens(1) in this case) is the starting point for receiving the dimensional information. Here are some examples of what will end up in num, dimens(1), and dimens(2) for various student responses:

student response	num	dimens(1)	dimens(2)
13.6 grams/cm^3	13.6	1	-3
13.6	13.6	ø	ø
13.6 cm-gm^2	13.6	2	1
13.6 kg/1øcm	136ø	1	-1

Note in the third example that a minus sign preceding a unit name is taken as a dash meaning multiplication, not subtraction. Note in the last example that "kg" brings in a factor of 1øøø relative to the basic unit (gm). Note also that, as usual, TUTOR does multiplications before doing divisions so that the "1ø cm" is all in the denominator, with the result that we have (length)$^{-1}$. Similarly, "1/2 kg" will be taken to mean 1/(2 kg), not (1/2) kg. As mentioned earlier, it is best to point out this matter to the student at the beginning of the lesson.

Like -store-, the -storeu- judging command will flip TUTOR to the regular state (with a "no" judgment) if it cannot evaluate the student's response. The system variable "formok" can be used in a -writec- to tell the student why his response can't be evaluated. One example peculiar to responses involving units is "5 grams + 3 cm", which is absurd. You cannot add masses and lengths, and -storeu- will give up. On the other hand, the student can say "65 cm + 2 meter" and -storeu- will set num to 265, dimens(1) to ∅ (no mass), and dimens(2) to 1. As another example, "cos(3cm)" is rejected, but "cos(3cm/meter)" is accepted. The argument of most functions must be dimensionless. (Exceptions are "abs" and "sqrt".)

A related difficulty faces students unless you explicitly warn them: "3+6 cm" is rejected by -storeu- although it looks reasonable in context to the human eye. As far as -storeu- is concerned, however, the student is trying to add 3 "nothings" to 6 cm, and the units do not have the same dimensionality. For -storeu- this is as improper as "3 kg + 6 cm". Unfortunately, until -storeu- (and TUTOR) get much smarter, it will be necessary to give explicit instructions to the students that

1) Multiplications are done before divisions (unless parentheses intervene), so that 1/2 kg does not mean (1/2) kg.

2) Responses such as "3 + 6cm" must be written rather as "(3+6)cm".

Note that these rules also apply in scientific journals and almost all textbooks, but your students may not be consciously aware of these standard rules. Given only these standard conventions, -storeu- will correctly handle an enormous variety of student responses.

While -storeu- can be used to get the number and dimensionality, usually the -ansu- and -wrongu- commands are used to check for specific cases. Let us modify our sample unit to use these commands, which are like -ansv- and -wrongv- except for checking for correct units:

```
:
:
arrow   1618
storeu  num,dimens(1)
write   Cannot evaluate!
ansu    13.6 gm/cm3,.1
write   Good!
wrongu  13.6,.1
write   Right number, but give the units!
wrongu  (num)gm/cm3,.1
write   Right dimensionality, but wrong number!
wrongv  13.6,.1
write   Right number but wrong dimensionality.
no
writec  dimens(2)=-3,Length ok.,Length incorrect.
```

The -ansu- will make a match only if the dimensionality is correct and the number is 13.6 plus or minus the tolerance (given as .1). The first -wrongu- checks for 13.6 $(mass)^{\emptyset}(length)^{\emptyset}$; that is, no units given at all. The second -wrongu- looks for a number equal to (num), which it surely finds because that is the number the student gave (as determined by -storeu-). Therefore, this -wrongu- will match if the number is not 13.6 but the dimensionality is correct. The -wrongv, unlike -wrongu-, doesn't care about dimensionality but only about the numerical part. It is used here to check for responses such as "13.6 cm".

The -exact- and -exactc- commands: a language drill

It is occasionally useful in special cases to use a command less powerful than -answer- to judge a response. Suppose you are teaching the precise format required on some business form, and you want the student to type "A   B   C" exactly, with three spaces between the letters. A match to "answer A   B   C" would occur no matter how the student separates the letters. One space, four spaces, a comma or a semicolon--any of these punctuations are permissible separators as far as -answer- is concerned. Normally this flexibility is good, for it keeps students from getting hung up on petty details. If, however, it is precisely the details that matter on a particular response, use an -exact- command. In the present case, the statement "exact   A   B   C" will be matched only if the student types exactly that string of characters: A, space, space, space, B, space, space, space, C.

At the time of writing the -answer- command does not permit punctuation marks in its tag, so that a response such as "a:b" must be judged with an -exact- command if the colon is important. While punctuation marks cannot appear in the tag of the -answer- command, the student can use them in a response. The -answer- command will treat all punctuation marks that the student uses as being equivalent to spaces.

It should be emphasized that it is easy to misuse the -exact- command. The student should normally be given considerable latitude in the form of his response, such as is permitted by -answer-, -concept-, and -ansv- commands. The -exact- command should be used sparingly, and only for rather short responses. If it is important that the student know the exact format of something as long as

3 No. 6 screws/516-213-86xq-4: New Orleans

it would certainly be preferable to have him pick this correct form out of a displayed set of samples than to ask him to type it exactly! Then all he need say is that item number 3 is the correct form.

There is a conditional -exact- command, -exactc-, which may be used to create simple vocabulary drills if the power of the -answer- command is not needed:

```
unit    espo
next    espo
at      1812
write   Give the Esperanto for
randu   item,5
at      2015
writec  item-2,one,two,three,four,five
arrow   2113
exactc  item-2,unu,du,tri,kvar,kvin
```

You might write yourself a similar unit to drill yourself on historical dates, capitals of nations, etc. This drill has three defects--it never ends; you may see the same item two or three times in a row; and no help is available if you get stuck. Let's revise it to have the following characteristics: It should present the five items in a random order but without repeating any item; any items missed will then be presented again; the student may press HELP to get the correct answer.

We will be using a random sequence of non-repeating item numbers such as

4,2,1,5,3.

This is called a "permutation" of the five integers. Another, different, permutation is the sequence

2,5,3,1,4.

You can see that there is a large number (120) of different permutations of five integers. Correspondingly, there is a large number of different permutation sequences for presenting our drill to the student. Such sequences of non-repeating integers are quite different from the sequences we get from repeated execution of our "randu item,5", which produces sequences such as

3,2,4,4,1,5,1,2,4,3,5,5,2,etc.

with some integers repeating and some not showing up for a long time.

We need some way of asking TUTOR to produce a permutation for us, rather than the kind of sequence produced by -randu-. This is done by telling TUTOR to set up a permutation of 5 integers ("setperm 5") from which to draw integers ("randp item") until the sequence is finished (indicated by "item" getting a value of zero). The -setperm- command actually sets up two copies of the permutation, and the "remove item" statement can be used to remove an integer from the second copy. (-randp- draws integers from the first copy.) If we -remove- only those integers corresponding to items correctly answered on the first try, the second copy will contain only the difficult items after completing the first pass over the five items. Then we can use -modperm- (which has no tag) to modify the first copy by shoving the second copy into the first copy. Having replenished the first copy with the difficult items we can use -randp- to choose these again.

Here is a form of the drill incorporating these ideas:

```
unit    begin
setperm 5                   $$ set up two copies of a permutation
jump    choose
*
unit    choose
calc    attempt⇐0           $$ initialize number of attempts
randp   item                $$ pick an integer
jump    item>Ø,espo,x       $$ fall through if first copy empty
modperm                     $$ use second copy
randp   item
jump    item>Ø,espo,x       $$ fall through if second copy empty
at      2115
write   Congratulations!
        You finished the drill.
end     lesson              $$ end the lesson
*
unit    espo
next    choose
help    esphelp
at      1812
write   Give the Esperanto for
at      2Ø15
writec  item-2,one,two,three,four,five
arrow   2113
exactc  item-2,unu,du,tri,kvar,kvin
goto    attempt>Ø,q,x
remove  item                $$ remove item if correct on first attempt
no
calc    attempt⇐attempt+1
*
unit    esphelp
calc    attempt⇐attempt+1  $$ count HELP as an attempt
at      1613
writec  item-2,unu,du,tri,kvar,kvin
end
```

We want to remove an item only if the student gets it right on the first try, which means "attempt" should be zero. "goto attempt>Ø,q,x" means "goto a fictitious, empty unit "q" if attempt is greater than Ø, else fall through". If we fall through, we remove the item ("remove item"). We increment "attempt" on each try and also when help is requested so that if the student has to see the answer, the item is not removed and will be seen again. Note that he is required to type the correct response and cannot see this answer while he types, which gives him additional practice on the difficult items.

At the present writing there is no conditional -answer- command. If the spelling power of the -answer- command is desired in this drill, replace the -exactc- with a conditional -join- to attach one of five units containing -answer- commands. You must use -join-, not -do-, since it will be used in the judging state.

Stages in processing the -arrow- command

Let us summarize the several stages of processing involved when there is an -arrow- command.

Stage 1	The -arrow- command is executed: the arrow is displayed on the screen, and a marker is set to remember the unit and location within the unit of this -arrow- command. Regular processing continues until a judging command is encountered, at which point there is a wait while the student types a response.
Stage 2	The student presses NEXT or otherwise completes his response. TUTOR uses its -arrow- marker to start judging at the statement following the -arrow- command. Only judging commands are executed--all regular commands are skipped. Execution of a -specs- command sets a -specs- marker to remember the unit and location within the unit of this -specs- command.
Stage 3	Some judging command terminates judging, and successive regular commands are executed until a judging command is encountered, which ends this regular processing, even if we are several levels deep in -do-s. There is no "undoing". An -arrow- or -endarrow- will also halt this regular processing without permitting "undoing". (If no judging command terminates the judging phase, the end of the unit, an -endarrow-, or another -arrow- will end Stage 3 and make a "no" judgment.)
Stage 4	If the -specs- marker has been set, regular processing begins at the statement following the last -specs- command encountered. (The -specs- marker is cleared.) This processing terminates in the same way as the regular processing of Stage 3. If the judgment is not "ok", the -arrow- is not satisfied. The student must erase part or all of the response and enter a different response, which initiates Stage 2 again.
Stage 5	The <u>search</u> state is initiated if there is an "ok" judgment. TUTOR again uses the -arrow- marker to start processing at the statement following the -arrow- command, this time in a search for another -arrow-. Only -join-s are executed; all other commands, regular or judging, are skipped during this search state. If an -arrow- command is encountered,

TUTOR begins Stage 1 for this additional -arrow-. If an -endarrow- command is encountered, the search state ends and regular commands are processed. If neither -arrow- nor -endarrow- is encountered, the student can press NEXT to go on to the next main unit, having satisfied all the -arrow-s.

Written out in full this way, this all sounds rather complicated, but in most practical cases this structure turns out to be quite natural and reasonable. It is, nevertheless, useful to look at some bizarre or pathological cases to further clarify the various processing stages.

<u>Repeated execution of -join-</u>

First here is an example of the repeated execution of a -join- in regular, judging, and search states:

```
unit    multy
calc    i⇐∅
arrow   1514
join    i⇐i+1,ansdog
endarrow
at      2514
show    i
*
unit    ansdog
answer  dog
write   Bowwow!
```

The conditional -join- has only one unit listed, so we will always join unit "ansdog" no matter what value the expression (i⇐i+1) has. Upon first entering unit "multy", we do the -calc-, the -arrow-, and the -join-, all in the regular state. This terminates at the -answer- command to await a student response. Note that i is now 1, due to the assignment (i⇐i+1) contained in the conditional -join-. Suppose the student types "cat" and presses NEXT. TUTOR starts at the statement following the -arrow- and executes the -join- in the judging state (incrementing 1 to 2 in the process). No match is found for "cat", so the student must give another response. Suppose he now enters "dog". TUTOR again starts judging just after the -arrow- and again executes the -join- (thus incrementing i to 3). This time there is a match to "answer dog" which changes the state from judging to regular. The "write Bowwow!" is executed, and the end of unit "ansdog" causes TUTOR to "undo" back into unit "multy", where the -endarrow- signals the end of the statements associated with the -arrow-. Since we got an "ok" judgment, we are ready to search for any other -arrow-s that might be in unit "multy". We return to the -arrow- one last time, this time in the search state. The -join- is executed to see whether there is an -arrow- command lurking in unit "ansdog", with the incidental result that i gets incremented to 4. No -arrow- is found in unit "ansdog" and we "undo" into the -endarrow- command, which changes us from search state to regular state. The -at- and -show- are executed and we get "4" on our screen, due to the quadruple execution of the -join-.

Aside from illustrating some consequences of the processing rules, this example should emphasize that using the assignment symbol (⇐) in a conditional -join- may have unexpected results! Note that -join- is the only command with these properties, due to the fact that it is the only command executed in regular, judging, and search states. It is important that -join- be universally executed in this way so that you can join judging commands in the judging state and even -arrow- commands in the search state, not just regular commands in the regular state.

Judging commands terminate regular state

The rule that a judging command terminates the processing of regular commands is an important and general rule. We have seen that this must be true upon first encountering an -arrow-: the first judging command after the -arrow- makes TUTOR wait for a student response, since that judging command needs a response to work on. Let's see another instance of the rule:

```
        .
        .
        .
        arrow    1518
        answer   dog
        write    Bowwow
        wrong    cat
        write    Meow
        wrong    horse
        .
        .
        .
```

If the student says "dog", he gets a reply "Bowwow" and regular processing stops at the "wrong   cat" because -wrong-, a judging command, terminates the regular state. Similarly, if the student response is "cat", the statement "write   Meow" is the only regular statement which is executed. The judging commands delimit those regular commands associated with a match of a particular judging command. This delimiting effect is achieved because

1) regular commands are skipped in the judging state; and
2) the processing of regular commands ends whenever a judging command is encountered.

Now let's consider a slightly modified sequence:

```
        .
        .
        .
        arrow    1518
        join     dogcat
        write    Meow
        wrong    horse
        .
        .
        .
        unit     dogcat
        answer   dog
        write    Bowwow
        wrong    cat
```

Supposedly the "join   dogcat" will act as though the statements of unit "dogcat" were inserted where the -join- is, which should make this modified version equivalent to the earlier version. Indeed, the rule that a judging command terminates the processing of regular commands does make the two versions equivalent, as we will show. Remember in this discussion that -join- is the same as -do- except for the universal nature of -join-.

Suppose the student types "dog". We start just after the -arrow-, in the judging state. The -join- is executed and we find a matching "answer dog" which ends judging and puts us in the regular state. The "write Bowwow" is executed. Then we come to a -wrong- judging command which stops the regular processing and prevents "undoing"! Even though we are one level deep in -do-s, TUTOR will not "undo"; the "write Meow" which follows the "join   dogcat" will not be executed. What will happen is just what happens in the earlier version: we have an "ok" judgment, which causes the search state to be initiated at the -arrow- (there was no -specs-). Thus the two versions operate in identical manners because the -join- acts like a text-insertion. Note that a response of "cat" will get a reply "Meow" because there is no judging command following the "wrong   cat"; a normal "undo" is performed at the end of unit "dogcat".

This last example illustrates the importance of the rule "a judging command terminates the regular state". It is this rule which insures that -join- (or -do-) will act like a text insertion.

We saw in the discussion of the -goto- command in Chapter VI that a -goto- in a done unit destroys the strict text-insertion character of the -do-. That is true in the present context as well. Suppose we insert a -goto- in unit "dogcat" (any -goto- will do; we'll use a "goto   q"):

```
        unit     dogcat
        answer   dog
        write    Bowwow
        goto     q
        wrong    cat
```

The student enters "dog" and we do unit "dogcat" where the match to "answer dog" flips us from the judging to the regular state. The regular commands -write- and -goto- are executed. (Note that -goto-, like -do-, is only regular, whereas -join- is universal, being executed not only in regular but in judging and search states.) The execution of the -goto- prevents TUTOR from encountering the "wrong cat" which previously terminated the regular state. We have run out of things to do in unit "dogcat" and are one level deep in -do-s. TUTOR, therefore, "undoes" and executes the "write Meow" which follows the "join dogcat"! The student will see "BowwowMeow" on his screen. If, on the other hand, we replace the "join dogcat" with the statements contained in unit "dogcat" we would have

```
.
.
.
arrow    1518
answer   dog
write    Bowwow
goto     q
wrong    cat
write    Meow
wrong    horse
.
.
.
```

and a response of "dog" would merely cause "Bowwow" to appear on the screen, not "BowwowMeow".

We have again seen that a -goto- in a done unit can cause the -join- operation to behave differently from a text-insertion. We get different effects depending on whether we -join- such a unit or put that unit's statements in place of the -join- statement.

This property of the -goto- can sometimes be used with good effect. Occasionally you may want to "undo" despite the presence of a following judging command, in which case you can use a -goto- to prevent TUTOR from seeing that judging command. Conversely, you can place a judging command at the end of some regular commands for the express purpose of preventing the "undo".

-goto- is a regular command

Since the -goto- command is a regular command, it is skipped in the judging and search states. Here is an erroneous sequence of commands which illustrates the fact that the -goto- is skipped in the judging state:

```
        .
        .
        .
        arrow   1612
        goto    dogcat
        *
        unit    dogcat
        answer  dog
        write   Bowwow
        wrong   cat
```

When the -arrow- is first encountered, an arrow is displayed on the screen at 1612. TUTOR continues in the regular state and executes the -goto-. The -answer- in unit "dogcat" ends this regular processing to await the student's response. Suppose the student types "dog" and presses NEXT. TUTOR starts judging just after the -arrow-, skips the regular -goto- command, and finds no judging commands at all! The student's response gets a default "no" judgment. The -goto- should be replaced by a -join- so that unit "dogcat" will be attached in the judging state.

Similarly, here is an erroneous sequence to illustrate the fact that the -goto- command is skipped in the search state:

```
        .
        .
        .
        arrow   1612
        specs   bumpshift
        answer  dog
        goto    another
        wrong   cat
        *
        unit    another
        arrow   2514
        answer  wolf
```

The student responds to the first -arrow- with "dog" and matches the "answer dog", which switches the processing from the judging state to the regular state. The -goto- is executed, and in unit "another" we encounter an -arrow- command. This -arrow- command terminates the regular processing just as a judging command would. The -specs- marker was set, so we will now execute any regular commands following the -specs- command (there are none in this example). Since the student's response was "ok", the search state is now initiated. TUTOR starts at the "arrow 1612" looking for another -arrow- command. The -specs-, -answer-, -goto-, and -wrong- are skipped in the search state, and we come to the end of the unit without finding an -arrow-. Thus the -goto- did not succeed in attaching a second -arrow-. If the -goto- is replaced by a -join-, the "wrong cat" will be associated with the second -arrow- (2514)! This is due to the text-insertion nature of the -join-, which interposes the statements of unit "another" between the "answer dog" and the "wrong cat". One correct way to write this sequence is like this:

```
        .
        .
        .
        arrow    1612
        specs    bumpshift
        answer   dog
        wrong    cat
        endarrow
        goto     another          $$ or "do     another"
        *
        unit     another
        arrow    2514
        answer   wolf
```

The -goto- or -do- placed after the -endarrow- will not cause any mischief because the search state has been completed: the -endarrow- flips us from search state to regular state.

Considerations of this kind mean that some care must be exercised when using -join- or -do- to attach units containing -arrow- commands. To avoid unpredictable results follow these two rules:

1) A unit attached by -join- or -do- which contains one or more -arrow- commands must end with an -endarrow- command. This insures that the unit will end and "undo" in the regular state. (It is permissible to have regular commands following the -endarrow-.)

2) The attached unit containing one or more -arrow- commands must not contain any -goto- commands. (A -goto- can make TUTOR fail to see the -endarrow- or a judging command so that a premature "undo" occurs.)

If these two rules are followed, the -join- or -do- will act precisely as though you had inserted the statements of the attached unit where the -join- or -do- was.

Interactions of -arrow- with -size-, -rotate-, -long-, -jkey-, and -copy-

When an -arrow- command is performed, several things happen. An arrow character is displayed on the screen, cuing the student to enter a response. A note is made of the unit and location within that unit of the -arrow- command so that TUTOR can return to this marked spot when necessary. Even the trail of -do-s (and/or -join's) which brought TUTOR to this -arrow- command is saved, so that each restart at the -arrow- will be at the appropriate level of -do- relative to the main unit. The current settings of -size- and -rotate- are saved to be restored each time, so that you can write a size-3 reply to a student's incorrect response without affecting the size of his corrected typing. In other words, response-contingent settings of -size- and -rotate- are temporary, whereas in other circumstances they are permanent until explicitly changed:

```
          .
          .
          .
          size      2
          rotate    Ø
          arrow     1718
          answer    dog
          size      4
          rotate    3Ø
          write     Woof!
          answer    wolf
          endarrow
          at        2218
          write     This is in size 2, rotate Ø.
          .
          .
          .
```

The last writing appears in size 2, rotate Ø despite the size 4, rotate 3Ø, that were contingent on the student's response, "dog". When the search state is initiated, the original size and rotate settings are restored. Similarly, if "dog" had been judged wrong, the student's revised typing would have been in size 2, not 4, because the original size and rotate are restored before waiting for the student's revised input.

Executing an -arrow- command has other important initialization effects:

1) A default response limit of 15Ø characters is set. The student cannot enter a response longer than 15Ø characters (including "hidden" characters such as shift-codes and superscripts). This can be altered by following the -arrow- command with a -long- command to change this to as much as 3ØØ. If this is a "long   1", judging will commence as soon as the student types one character. If more than 1 is specified the student is prevented from entering more characters but must press NEXT to initiate judging, unless a "force   long" statement has appeared in the unit.

2) A default specification of "judging keys" is set. Only the NEXT key will cause judging to start (or one character if there is a "long 1", or hitting the limit with a "force long"). This can be altered by following the -arrow- command with a -jkey- command to specify additional judging keys (NEXT is always a judging key). An example is "jkey data,help" which would make the DATA and HELP keys equivalent to the NEXT key at this arrow.

3) A default specification is set to disable the COPY key. The -arrow- command can be followed with a -copy- command to specify a previously-stored character string to be referenced with the COPY key. An example is "copy v51,v3", where v51 is the start of the character string and v3 is the number of characters. This way of specifying a string of characters is the same as the scheme used with -storea- and -showa-.

Some explanation of the COPY and EDIT keys is required. The EDIT key is always available for the student to use in correcting his typing. Pressing the EDIT key the first time erases all typing, after which each press of the EDIT key brings back the typing one word at a time. This makes it easy to make corrections and insertions without a lot of retyping. Each press of the COPY key, on the other hand, brings in a word from the character string specified by the -copy- command, as opposed to bringing in the student's own typed words with the EDIT key. One example of the use of the COPY key is seen in the PLATO lesson editor, where you as an author can use the COPY key in insert or replace mode to bring in portions of a preceding line without having to retype. The COPY key must be specifically enabled by a -copy- command, but the EDIT key is always usable, unless you specify a -long- greater than the normal limit of 15Ø. (To use the EDIT key on responses longer than 15Ø characters requires you to furnish an edit buffer through an -edit- command.)

The -long-, -jkey-, and -copy- commands all override default specifications set by the -arrow- command. They can be thought of as modifiers of the -arrow- command. If they are to have an effect on the student's first response, they not only must follow the -arrow- command but must precede any judging commands:

```
    .
    .
    .
    arrow   1518
   {jkey    help
   {copy    cstring,ccount
   {long    15
    -specs- or -answer- or -store- or any other judging command
```

If -jkey-, -copy-, or -long- came after the first judging command, the -arrow- defaults would hold for the first response because the modifying command would not have been executed yet.

Applications of -jkey- and -ans-

Use of the -jkey- command is well illustrated in the case of providing help to the student through the HELP key but without leaving the page. If giving help requires an entire screen display, or a whole sequence of help units, it is best to use a -help- command to specify where to jump if the student presses HELP. The screen is then erased automatically to make room for the help page (unless the original base unit had an "inhibit erase" in it). On the other hand, the appropriate help might consist merely of a brief comment or some additional line-drawings on the present page. A convenient way to provide such help without leaving the page is this:*

```
        .
        .
        .
        arrow   1815
        jkey    help
        answer  cat
        no
        write   Hint: it meows...
```

The statement "jkey help" makes the HELP key completely equivalent to the NEXT key. If the student presses HELP, judging is initiated, his (blank) response does not match "cat", and he gets "Hint: it meows...." Without the -jkey- command, the HELP key would be ignored which would be bad. It is a very good idea to have the HELP key do something at all times so that the student can come to expect help to be available.

In this example the student will get the same assistance whether he presses HELP or whether he types "dog" followed by pressing NEXT. We could give different kinds of assistance in these two cases by changing the -write- statement to a -writec-:

```
        .
        .
        .
        arrow   1815
        jkey    help
        answer  cat
        no
        writec  key=help,Meow?,The answer is cat.
```

The system variable "key" always contains a number corresponding to the last key pressed by the student. In this case the last key will either be HELP or NEXT. If the student presses HELP, the logical expression "key=help" will be true (-1) and the student gets the reply "Meow?" But if he presses NEXT, then the logical expression "key=help" is false (Ø) and the student gets "The answer is cat." The lower-case word "help" is defined by TUTOR to mean in a calculational expression "the number corresponding to the HELP key". Other such defined names include next, back, help1 (for shift-HELP), etc.

* There is now a -helpop- command, similar to -help-, for providing help on the page (op) without erasing the screen.

An alternative way of writing the same sequence is this:

```
      .
      .
      .
      arrow   1815
      jkey    help
      no                          $$ terminate judging
      judge   key=help,x,continue
      write   Meow?
      answer  cat
      no
      write   The answer is cat.
```

If key=help, we "fall through" the -judge- command and write "Meow?" If the key is not equal to help (that is, the student pressed NEXT), a "judge continue" is performed to return to the judging state. The "write Meow?" is skipped since -write- is a regular command. If the response does not match "cat", the student will get the message "The answer is cat". As usual, there are many equivalent ways in TUTOR to do the same thing! In a particular situation one scheme may be more appropriate than another.

There is an ANS key on the keyset which is often used to let students skip through material by just pressing ANS:

```
      .
      .
      .
      arrow   1817
      jkey    ans
      ok
      judge   key=ans,x,continue
      write   The answer is cat
      answer  cat
      .
      .
      .
```

Since the ANS key generates an ok judgment here, the student will move on immediately to the next arrow or unit without having to type the correct answer. This could be made conditional on the student being in a review mode. That is, you might define "review=v1", zero it initially, and set it to -1 only after the student has gone through the material once under his own power. With the following structure

```
        .
        .
        .
        arrow    1817
        do       review,jans,x
        ok
        judge    key=ans,x,continue
        .
        .
        .
        unit     jans
        jkey     ans
```

he will be able to use the ANS key only when reviewing the material.

There is an -ans- command (blank tag) which is equivalent to

```
        jkey     ans
        ok
        judge    key=ans,x,continue
```

In other words, you just write

```
        .
        .
        .
        arrow    2123
        ans
        write    The answer is cat.
        .
        .
        .
```

The -ans- command is a judging command and must be the first judging command after the -arrow-. When it is first encountered, it sets up ANS to be a judging key, and it is matched only if the ANS key is pressed. If the -ans- command is used only to provide a kind of help, but not to let the student pass on to the next thing, put a "judge wrong" after the -ans- command.

In many places you may do specific things in response to the ANS and HELP keys. Elsewhere in the lesson it is appropriate merely to enable them so that something will happen when these keys are pressed. Just put "jkey help,ans" after each such -arrow-. The student will then get at least whatever reply you give him after the universal -no- that catches all unrecognized responses. Certainly every -arrow- should provide some kind of feedback to unrecognized responses or the student will get hung up. The "jkey help,ans" will further insure that a reasonable response to his input is always forthcoming. Without this -jkey- statement, nothing would happen when the student presses ANS or HELP.

An additional refinement is advisable. Often a student will press NEXT an extra time, perhaps because he hadn't noticed that he was to type a response. This blank response, consisting of just a NEXT key, will probably get judged "no" at most arrows, which requires an additional NEXT (or ERASE) to clear the "no" judgment before typing a response. This can get confusing. In most cases it is best simply to ignore blank responses:

```
     .
     .
     .
     arrow    1914
    {exact                    $$ check for blank response
    {judge    ignore
     answer   cat
```

This has the effect of throwing away superfluous NEXT keys.*

On the other hand, you should accept blank responses involving ANS or HELP. It is useful to write

```
     .
     .
     .
     arrow    1917
     join     anshelp
     answer   cat
     .
     .
     .
     unit     anshelp
     jkey     ans,help
     exact
     judge    key=next,ignore,continue
```

Placing "join   anshelp" after each -arrow- will insure that extra NEXT keys are thrown out but that responses involving ANS or HELP keys will fall through to whatever reply you give to unrecognized responses. Note that you must use -join-, not -do-, to attach unit "anshelp", since you want not only to do unit anshelp in the regular state to specify -jkey- before the student responds. When the student responds, you also want to do unit anshelp in the judging state in order to reach the -exact- judging command.

Modifying the response: -bump- and -put-

It is possible to delete characters from the judging copy of the student's response by using the -bump- command:

```
     .
     .
     arrow    1812
     bump     as3               $$ delete all a's,s's, and 3's
     answer   rdvrk
```

* The statement "inhibit blanks" can now be used to ignore superfluous NEXT keys.

This -answer- will be matched if the student types "33 aardvarks" because the -bump- command reduces the judging copy of response to " rdvrk". The original response is not altered and can be recovered with a "judge rejudge". Also, the screen display is unaffected: the student still sees on his screen "33 aardvarks", just as he typed it. On the other hand, all judging commands following the -bump- are affected since they all operate on the judging copy, not on the original response. For example, a -storea- following the -bump- would give you "rdvrk". Here is another silly example:

```
define   cfirst=v1,csecond=v2
         first=v11,second=v21
unit     conson
at       913
write    Type anything, and I'll
         remove the vowels:
arrow    13Ø9
long     1ØØ                      $$ from v11 to v21 is 1ØØ characters
storea   first,cfirst⇐jcount
bump     aeiou
storea   second,csecond⇐jcount
ok
write    You typed ◁a,first,cfirst▷
         Remove vowels: ◁a,second,csecond▷.
         You used ◁s,cfirst-csecond▷ vowels.
```

Note that "cfirst" is the number of characters (including hidden characters such as shift characters) in the original response, whereas "csecond" is the number of characters after the -bump- has removed the vowels. This is true because "jcount" always has an up-to-date character count of the judging copy, as influenced by -bump- and related operations. (You may recall that "specs bumpshift" also affects "jcount" by removing shift characters.) Suppose the student types "Apples taste funnier". Then he will get this reply:

```
You typed Apples taste funnier.
Remove vowels: Ppls tst fnnr.
You used 7 vowels.
```

The reason that the word "Apples" turns into "Ppls" with a capital "P" is that a capital "A" is really a shift character followed by a lower-case "a". With the "a" bumped out, the shift character stands next to the "p", making a capital "P".

While the -bump- command will delete characters, the -put- command will change particular strings of characters:

```
        .
        .
        .
        arrow   1218
        put     cat=dog
        put     rat=mouse
        storea  first,cfirst⇐jcount
        ok
        showa   first,cfirst
```

All occurrences of "cat" change into "dog", and all occurrences of "rat" change into "mouse". Suppose the student types "Scattered cats scratch rats". The reply will be "Sdogtered dogs scmousech mouses"!

Both -bump- and -put- are judging commands: they operate on the student's response. Like all judging commands, they stop processing when encountered during the processing of regular commands. The -put- command has a property similar to -store- in that it can terminate judging with a "no" judgment if it cannot handle the student's response:

```
        .
        .
        .
        arrow   1218
        put     cat=enormous
        write   Too many cats!
        ok
        .
        .
        .
```

If the student has many "cats" in his response, the -put- may cause "jcount" to exceed the 300-character response limit. In this case, it changes to the regular state, and the student gets the message "Too many cats!" This regular -write- command normally is skipped, since we're in the judging state.

There is an equivalent form of -put- which is often easier to read:

```
        put     cat=dog
        putd    /cat/dog/
        putd    ,cat,dog,
```

All three of these statements are equivalent. The -putd- (d for delimiter) takes the first character as the delimiter between the two character strings. Other examples of its use are these:

```
        putd    /=/equals/          $$ convert = sign
        putd    / //                $$ remove all spaces
```

It is also possible to change variable character strings by using -putv- (v for variable):

```
        putv    first,cfirst,second,csecond

          string and count        string and count
```

In using these -put- and -bump- commands in combination, care must be taken in their order. For example, the following sequence is nonsense:

```
        bump    a
        put     cat=dog
```

With all a's bumped the -put- will not find any cat's. Similar remarks apply to sequences of -put- commands.

The -bump- command looks for single characters, so "bump   B" will not merely bump capital B's. All shift characters will be bumped as well as lower-case b's. In other words, "bump   B" is really "bump   shift-b". If you want to eliminate only capital B's, use "putd   /B//". This will find occurrences of the string of characters "shift-b" and replace this string with a zero-length string, thus deleting the B.

The main purpose of -bump- and -put- is to make minor modifications to the student's reponse to convert it into a form which can be handled by standard judging commands. For example, the word-oriented judging commands (-answer-, -match-, -concept-, etc.) cannot find <u>pieces</u> of words. Suppose that for some reason you need to look for the fragment "elect"; you don't care whether this appears in the word "selection" or "electronics" or "electoral". Do this:

```
        •
        •
        •
        arrow   1723
        specs   okextra
        putd    /elect/ elect /
        answer  elect
        •
        •
        •
```

The -putd- is used here to put spaces before and after the string "elect" so that it stands out as a separate word. You could also use the values of "jcount" before and after executing the -putd- to determine whether "elect" was present. How many times it appeared could also be determined from these values. The value of "jcount" will increase by two for each insertion of two extra spaces.

Manipulating character strings

The judging commands -bump- and -put- operate on the judging copy of the student's response. It is sometimes useful to manipulate other strings of characters with -pack-, -move-, and -search-. These commands are regular commands, not judging commands; like -showa-, they operate on stored character strings, not the judging copy of the student's response. These commands are mentioned here because they are often used in association with analyzing student responses. In particular, the judging command -storea- can be used to get the response character string. Then it can be operated on with -move- and -search-. Finally the altered character string can be loaded back into the judging copy with the judging command -loada- (load alphanumeric; the -loada- command is precisely the opposite of -storea-). Since this section deals with a rather esoteric topic, you ~~might~~ just skim through it now to get a vague idea of what character string manipulations look like. If later you find a need for such operations, you should return to study this section with some care.

Here is an example of a -move- statement:

```
          move    v3,5,v52,21,8
```

This means "move 8 characters from the 5th character of the string that starts in v3 to the 21st character of the string that starts in v52". The 21st through 28th characters of the v52 character string are replaced by the 5th through the 12th characters of the v3 character string. The v3 character string is unaffected. In other words, -move- has the form

```
          move    string1,start1,string2,start2,#characters moved
```

If the number of characters to move is not specified, one character will be moved.

Here is an example of the use of -move-. Suppose the student types "x+4y = y-3" and we want to convert this into the form "x+4y-(y-3)" before using -store- on it. Assume "str" has been defined:

```
          .
          .
          .
          arrow   1812                    $$ x+4y=y-3
          putd    .=.-(.                  $$ x+4y-(y-3
          storea  str,jcount
          ok                              $$ to do regular -move-
         {move    ')',1,str,jcount+1      $$ x+4y-(y-3)
         {judge   continue                $$ to do judging -loada-
         {loada   str,jcount+1
          store   result
          ok
          write   Subtracting the right side of
                  your equation from the
                  left side gives ⊲s,result⊳.
```

In the -move- command the parenthesis within single quote marks, ')', means a character string one character long consisting of a right parenthesis. Similarly, 'dog' would denote a character string consisting of d,o, and g. Character strings up to ten characters in length may be described this way, using single quote marks. The -move- command shown above moves the ~~first~~ character of ')', which is just a right parenthesis, to the (jcount+1)th character position in "str". This effectively appends a right parenthesis to the student's character string (as modified by the -putd-). The -loada- command moves the final character string into the judging copy so that -store- can operate on it. Note carefully the switches from judging state to regular state and back again.

The -search- command is used to look for occurrences of specific character strings. It has the form

$$\text{search}\ \ \underbrace{\text{string1,length1}}_{\text{string sought}},\underbrace{\text{string2,length2}}_{\substack{\text{string to}\\\text{look through}}},\underbrace{\text{start2}}_{\substack{\text{where to}\\\text{start}}},\underbrace{\text{return}}_{\text{return location}}$$

Suppose we use -storea- to place the unaltered student response "x+4y=y-3" in "str,jcount". Then use

$$\text{search}\ \ \underbrace{\text{'=',1}}_{\substack{\text{look for}\\\text{= sign}\\\text{(string 1}\\\text{character}\\\text{long)}}},\underbrace{\text{str,jcount}}_{\substack{\text{string}\\\text{to look}\\\text{through}}},\underbrace{\text{1}}_{\substack{\text{start at}\\\text{beginning}\\\text{of string}}},\underbrace{\text{charnum}}_{\text{return location}}$$

This -search- command will set the variable "charnum" to 5, since the equal sign is the 5th character in "x+4y=y-3". If the search is unsuccessful, "charnum" is set to -1. As further illustration of -move- and -search-, let's rewrite our earlier sequence without the -putd-:

```
arrow   1812
storea  str,jcount
ok
search  '=',1,str,jcount,1,charnum
* Now make room for the -( :
move    str,charnum+1,str,charnum+2,jcount-charnum
*Next insert the -( :
move    '-(',1,str,charnum,2    $$ move 2 characters
* Append the ) :
move    ')',1,str,jcount+2
judge   continue
loada   str,jcount+2
store   result
ok
```

The -search- finds the equal sign. The first -move- moves the latter part of the string to make room for the insertion of '-('. The second -move- makes the insertion which overwrites the characters (=y) which were there originally. The third -move- appends the ')'. Normally the -search- would be followed by a "goto charnum,noeq,x" to take care of the case where the student did not in fact use an equal sign, in which case "charnum" would be -1.

The single quote marks can be used to specify character strings up to ten characters long. Longer character strings can be placed in variables with a -pack- command:

```
        pack     v11,abcdefghijklmnopqrstuvwxyz
```

This packs a character string 26 characters long into v11 and following variables. Since each variable holds ten characters, v11 and v12 will be full, while v13 will have the last six characters. The -pack- command might be considered analogous to -storea-, since both place character strings in variables. In the case of -storea- the total character count can be gotten from the system-defined variable "jcount". The character count involved in a -pack- statement can be obtained with an optional form:

```
        pack     v3,v11,abcdefghijklmnopqrstuvwxyz
    character count    string location
```

In this case, the -pack- command will set v3 to 26, the total character count. This form of the -pack- statement is the only exception to the general TUTOR rule that optional parts of a statement go at the end of the statement. This exception is due to the unique string syntax of the -pack- command. It does have the effect of turning some things around:

```
        pack     v1,v2,H₂SO₄
        showa    v2,v1
```

This will display "H_2SO_4" on the screen. Note the inversion of order of v1 and v2 in the two commands. Incidentally, the character count in v1 will be ten, including three shift codes and two subscripts. The character string H_2SO_4 is actually composed of shift, h, subscript, 2, shift, s, shift, o, subscript, 4.

There are a few other string-oriented commands: -clock- will get the time, -date- gets today's date, -name- gets the (18-character) name the student is registered under, and -course- gets the course he is registered in. For example:

```
        .
        .
        .
        name    v1                        $$ v1 and v2 for name
        course  v3
        clock   v4
        date    v5
        write   Hello!  Your name is <a,v1,18>.
                You are registered in <a,v3>.
                The time is <a,v4>
                The date is <a,v5>.
```

Suppose the student is registered as "sam nottingham" in a course "french4". It is 10:45:37 PM (22:45:37 on a 24-hour clock) on June 3, 1974. The student will receive this display:

```
Hello! Your name is sam nottingham.
You are registered in french4.
The time is 22.45.37.
The date is 06/03/74.
```

All of these commands, -name-, -course-, -clock-, and -date-, simply place the requested character string in the specified variable for use in a -showa-.

The -clock- command produces a <u>character string</u>. There is a system variable "clock" which may be used in calculational expressions: it holds the <u>number</u> of seconds of a daily clock to the nearest thousandth of a second. It is convenient for calculating the amount of time spent in a section of a lesson.

The -date- command produces a <u>character string</u>. The -day- command produces a <u>number</u> corresponding to the number of days elapsed since January 1, 1973. This number of days and fraction of a day is accurate to one-tenth of a second.

The TUTOR judging commands offer a great deal of power. We have seen that the judging commands -bump- and -put- together with the regular string-oriented commands -move-, -search-, and -pack- can be used to change an otherwise intractable response into a form which can be handled with TUTOR judging commands. This is a useful scheme as long as only minor modifications are required. However, if major modifications of the response are required in order to be able to use TUTOR judging facilities, it is usually simpler to "do your own judging". That is, just get the student's response with a -storea- and then analyze it with string-oriented commands, together with the additional calculational machinery described in chapter IX. You might not even want to use the built-in marker features of the -arrow- command, with the associated returns to the -arrow- when there is a "no" judgment. In such circumstances you might write a subroutine to be used in place of -arrow- commands, which merely collects the student's response:

```
unit      arrow(apos)
arrow     apos
storea    sstr,scnt⇐jcount
specs     nookno
ok
endarrow
```

Instead of writing "arrow 1815" with associated judging commands you would then write

```
.
.
do        arrow(1815)
calc,move,etc. to do your own judging
.
.
```

Naturally this course of action is advisable only if you are trying to analyze responses which have a form very different from those classes of responses which can be handled well by TUTOR judging commands.

Catching every key: -pause-, -keytype-, and -group-

Sometimes it is useful to process individual keypresses without waiting for a NEXT key. We have already discussed such typical examples as moving a cursor and choosing a topic from an index. These examples used a "long 1" with an -arrow- in order to catch each keypress. There is another way to do this, involving the -pause- command which was introduced in chapter II in connection with creating displays, particularly timed animations. As was pointed out in the discussion of the -jkey- command in the present chapter, the system variable "key" contains a number corresponding to the most recent key pressed by the student. For example, if the student presses the letter "d", the system variable "key" will have the numerical value 4 (since d is the 4th letter in the alphabet). Putting these notions together, we have the following kind of structure:

```
write     Press "d", please.
pause
writec    key≠4,You didn't press d.,Good!
```

The blank -pause- statement ("blank" in the sense of having no tag) causes TUTOR to wait for the student to press a key. Any key will cause TUTOR to move past the -pause- to the next statement.

In the example shown, the -pause- is followed by a -writec- conditional on "key≠4". This -writec- can be written in more readable form by replacing the "4" by "d":

```
        writec  key≠"d",You didn't press d.,Good!
```

Enclosing the d with (double) quote marks is taken in calculational expressions to mean the number 4. Similarly, (v3⇐"z") will assign the value 26 to v3. If the student presses Ø or 1, "key" will have the numerical value 27 or 28 respectively. That is, the 26 letters are followed by the numbers Ø through 9, then come various punctuation marks. If the student presses the plus key, "key" will have the numerical value "+", which happens to be 37. If the student presses a capital D, "key" will have the value 64+"d", or 68. The shifted or upper case letters have "key" value 64 greater than the corresponding lower-case letters. Caution: some common keys such as parentheses have key numbers smaller than 64 despite requiring the shift key to type them. The most commonly used characters (lower-case letters, numbers, and common punctuation marks) have key numbers less than 64, independent of whether they are typed using the shift key. As for the function keys (NEXT, BACK, HELP1, etc.), we already saw in connection with the -jkey- command that the corresponding key numbers are given by next, back, help1, etc., as in

```
        goto    key=help1,yes,no
```

No quote marks are used for the function keys.

A more convenient way to determine which key has been pressed is to use a -keytype- command. Consider a cursor-moving procedure:

```
        define  num=v5,x=v1,y=v2,dx=1Ø,dy=1Ø
        unit    cursor
        pause
        keytype num,d,e,w,q,a,z,x,c
        goto    num,cursor,x
        .
        .
        calcs   num-1,y⇐y,y+dy,y+dy,y+dy,y,y-dy,y-dy,y-dy
```

The -keytype- command searches through the listed keys (d, e, w, q, a, z, x, and c in this case) and, similar to the -match- command, sets "num" to -1 if the key is not found in this list, or to Ø, 1, 2, 3, etc., if it is found. If the student presses d, "num" will be set to Ø; if he presses c, "num" will be 7; and if he presses D, "num" will be set to -1. The -goto- statement effectively causes all unlisted keys to be ignored.

Note that no quote marks are used in specifying keys in a -keytype- command. Capital letters and function keys may also be listed:

```
        keytype v3,a,A,b,B,next,data,timeup
```

While the -keytype- command is most often used in conjunction with a -pause- command, it can be used in association with an -arrow- command or any time that you want to decide what key was pressed most recently. The function key timeup is one generated by TUTOR when a timing key is "pressed" as the result of an earlier -time- command (see chapter II).

Just as the -list- command can be used to specify a set of synonomous words and numbers for use in -answer- and -match-, so there is a -group- command available for specifying synonomous keys for use in a -keytype- command:

```
        define  keynum=v23,algkey=v24
        group   algebra,x,y,z
        .
        .
        .
        keytype keynum,a,b,algebra,help
                       ↓ ↓    ↓      ↓
                       Ø 1    2      3
```

If the student presses any of the keys x, y, or z, the variable "keynum" will be assigned the value 2. An additional -keytype- command can be used to separate members of a group:

```
        keytype keynum,a,b,algebra,help
        goto    keynum,none,ua,ub,alg,somehelp
        .
        .
        unit    alg
        keytype algkey,x,y,z
        .
        .
```

Some particularly useful -group- definitions are built-in. Without specifying them with your own -group- commands, you can in a -keytype- command refer to these groups:

alpha	all 52 lower-case and upper-case letters
numeric	Ø through 9
funct	function keys (next,help,etc.)

An example of the use of these built-in groups might be "keytype v45,funct,a,b,c". You can also use previously defined or built-in groups in defining new groups:

```
        group   mine,a,b,c,help
        group   ours,mine,d,e,f
        group   all,A,B,C,ours,numeric,funct
```

It is important to note that if you use a -pause-, the key pressed will not cause the associated character to appear on the student's screen. You are in complete control. You may write something on the screen or not, as you choose. Only if you use an -arrow- will the standard key display take place, with the associated ERASE and other standard typing features available. Similarly, if you press HELP, you will not automatically branch to a unit specified by a previous -help- command, because a blank -pause- gives you every key, function key or not.

There is a variant of the -pause- command which is usually more useful than the blank -pause-. You can define which keys are to be accepted, and all other keys will be ignored:

```
        .
        .
        next    umore
        help    discuss
        data    tables
        .
        .
        pause   keys=d,D,next,term,help,help1
        .
        .
```

Any key not listed here is completely ignored, as though the student had not pressed it. Of the function keys listed, the HELP key will take the student to unit "discuss", since you have already specified what you want the HELP key to do. Note that this is not possible with a blank -pause- which catches all keys. Similarly, what the TERM key will do has been predefined: the student will be asked "what term?" But the DATA key will be ignored since it is not listed in the -pause- statement, and the student cannot reach unit "tables" with the DATA key until he has passed the -pause-. Pressing d, D, NEXT, or HELP1 will take the student past the -pause-. The NEXT key is slightly special here in that the preceeding specification "next   umore", unlike "help   discuss", tells TUTOR what to do when the present main unit has been completed. Thus pressing NEXT here just takes us past the -pause- rather than branching us immediately to a different unit as HELP does.

If you want the HELP key not to be ignored but not to access unit "discuss", the statement "help discuss" must follow the -pause- statement, or a "help q" must precede the -pause- in order to quit specifying a help unit.

Summary

This chapter has demonstrated an array of techniques for judging various types of student responses. There are -answer- and -wrong- (aided by -list-) for handling sentences composed from a rather small vocabulary of words. There is -concept- (supported by -vocabs-) to handle dialogs involving a large vocabulary. The -match- and -storen- commands can be used to pull out pieces of a student's response. There are -ansv-, -wrongv-, -ansu-, and -wrongu-, aided by "define student", for judging numerical and algebraic responses. The -exact- and -exactc- commands can be used when it is important that the response take a particular precise form. The -specs- command permits you to control various options associated with these commands and also provides a convenient marker of centralized post-judging processing. The regular -judge- command offers additional control over the judging process. The -bump- and -put- commands can be used to change a student's response into a form more easily handled by the standard judging commands. The -jkey- command can be used to cause judging to start when keys other than NEXT are pressed. The construction of randomized drills was illustrated, with the use of -setperm-, randp-, -remove-, and -modperm-.

You have also seen how to store numeric and alphanumeric responses for later processing (-store- and -storea-). These capabilities make it possible to "do your own judging" in those cases where the standard judging commands are not suitable. The basic TUTOR judging commands provide a great deal of power but cannot handle all possible situations. Fortunately there is always the possibility of performing calclations on stored student response, which means that TUTOR is open-ended in its judging power. The regular commands -search- and -move- can be used to manipulate stored character strings. (In chapter IX you will find discussions of "segments" and "bit manipulations" which permit you to use the -calc- command to perform additional operations on character strings.) We also discussed how to handle input from the student by collecting each key with a -pause- command, then using -keytype- (aided by -group-) to make decisions on a key-by-key basis.

We also discussed in some detail the marker properties of the -arrow- command. The -arrow- command serves as an anchor point which TUTOR clings to until the -arrow- is satisfied by an "ok" judgment, at which point a search is made for additional -arrow- commands. We looked at some cases involving the repeated execution of -join- in regular, judging, and search states, and of the non-execution of -goto- in the judging and search states. We also looked at other side-effects of the -arrow- command, including initializations associated with -size-, -rotate-, -long-, -jkey-, and -copy-.

It is planned that TUTOR have additional judging capabilities added to it in the future. One major improvement is imminent: the use of multi-word "phrases" such as "Santa Maria" and "out of order" in -answer- and related commands. Such phrases would be handled essentially as though they were single words. This turns out to be extremely useful in judging sentences correctly. This phrase feature is now available in -vocabs- but has not yet been extended to the other word-oriented TUTOR machinery.

It is hoped that you will re-read this chapter occasionally in the course of writing curriculum materials. The TUTOR judging capabilities are extremely rich because of the wide range of student responses that must be handled properly for lesson material to be successful. You should dip into this chapter occasionally as your own work suggests questions as to how to judge various classes of responses. Do not feel unhappy that some aspects of judging aren't clear to you at this time. Just be sure to reread appropriate sections of this chapter at a later time, when you need the details. For now it is sufficient to know what is available, and roughly in what form.

VIII. Additional display features

More on the -write- command

It should be pointed out that the -at- command not only specifies a screen position for subsequent writing but also establishes a left margin for "carriage returns" (CR on the keyset) in analogy to a typewriter. Upon completion of one line of text, the next line will start at the left margin set by the last -at- command. There are carriage returns implicit in "continued" write statements:

```
at      1215
write   Now is the
        time for all
        good men to
        come home.
```

The "at 1215" establishes a left margin at the 15th character position so that each line will start there. This example will produce an aligned screen display similar to the appearance of the tags of this continued -write- statement.

The setting of a margin by -at- has an unusual side effect. Consider:

```
at      2163
write   The cow jumped.
```

This will put the following display on the screen:

```
Th
e
co
w
ju
mp
ed
.
```

This is due to the left margin at character position 63, just two characters shy of the right edge of the screen. When a -write- would go past the right edge of the screen, TUTOR performs a carriage return to drop down one line, starting at the left margin. An -arrow- also sets a left margin with respect to the student typing a long response which would pass the right edge of the screen; further typing appears on the next lower line starting at the margin set by -arrow-.

It is important to understand that writing characters on the screen automatically advances the terminal's current screen position. Suppose we have consecutive -write- statements:

```
         at      712
         write   horses
         write   and cows
```

This sequence will display "horsesand cows" all on line 7. The first -write- ("horses") advances the terminal's screen position from the 712 specified by the preceding -at- to 712+6=718, there being 6 characters in the text "horses". Without an explicit -at- to change this, the second -write- ("and cows") starts at position 718. Note that

```
         at      712
         write   horses
                 and cows
```

would give a different display:

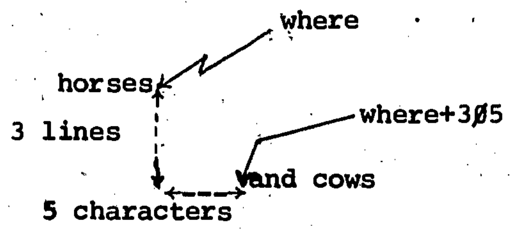
```
                 horses
                 and cows
```

because the "continued" -write- statement implies carriage returns.

TUTOR keeps track of the current screen position in a system variable named "where". For example,

```
         at      712
         write   horses
         at      where+3ø5     $$ "where" is 712+6=718 here
         write   and cows
```

will produce the display



The statement "write horses" leaves the screen position at 712+6=718, and the system variable "where" therefore has the value 718. When you then say "at where+3ø5" this is equivalent to saying "at 718+3ø5" or "at 1ø23".

There are many uses of this "where" system variable. Here is another example:

```
          at      1215
          write   What is your name?
          arrow   where+3
```

This will appear as

```
          What is your name? ⊳ Sam
```

The arrow has been positioned 3 characters beyond the end of the -write- statement's display.

The positioning information is useful with other display commands as well. Consider this:

```
          at      815
          write   Look at this!
          draw    where;815
```

This will display underlined text:

<u>Look at this!</u>

This is due to the fact that upon completion of the -write- statement "where" refers to the beginning of the next character position, just after the exclamation point. We simply draw from there back to the starting point. This form of the -draw- statement is so common that a concise form is permitted: "draw ;815" is equivalent to "draw where;815". Either form will draw a line or figure starting at the current screen position. This is particularly useful in constructing a graph by connecting the new point to the last point with a line. The point reached with a -draw- (or <u>any</u> display command) will be the new screen position and may be referred to through the system variable "where", which is kept up to date automatically by TUTOR.

There are fine-grid system variables "wherex" and "wherey" which correspond exactly to the coarse-grid "where". The position "where+3Ø5" is equivalent to "wherex+(5×8),wherey-(3×16)" because a character space is 8 dots wide and 16 dots high. The minus sign is present because in coarse grid line 4 is <u>below</u> line 3, whereas in fine grid dot 472 is <u>above</u> dot 471.

Superscripts and subscripts may be typed either in a locking or non-locking mode. To type "$1Ø^{23}$" you can either (a) press 1, press Ø, press SUPER, press 2, press SUPER, press 3 (non-locking case) or (b) press 1, press Ø, press shift-SUPER (that is, hold down the shift key while pressing SUPER), press 2, press 3. To get down from a locked superscript you type shift-SUB (locking subscript). Notice that in typing superscripts or subscripts the SUPER and SUB keys are pressed <u>and released</u> before typing the material to be moved up or down; you do <u>not</u> hold these keys down while typing, unlike the shift key used for making capital letters.

It is possible to overstrike characters to make combinations. The symbol "$\bar{v}$" can be made by typing v, backspace, SUPER, minus sign. This will superimpose a raised minus sign above the v. The backspace is typed holding down the shift key while hitting the wide space bar at the bottom of the keyset. Similarly, "horse" can be typed by typing "horse" followed by five backspaces and five underline characters. Note that these superpositions of characters won't work in "mode rewrite", where a new character is written on the screen. In mode rewrite the last example would show up as "_____", the "horse" having been wiped out by the characters whose only visible dots are the low, horizontal bars.

Extensions to the basic character set

We've seen examples of lower-case and upper-case characters, numbers, punctuation marks, superscripts, and subscripts. What if you need special accent marks, or an unusual mathematical symbol, or the entire Cyrillic alphabet for writing Russian? It is important that you be able to write text on the screen using the special symbols of your particular subject area. In addition, it is possible to use special characters to display small, intricate figures whose display would be slow and cumbersome if done with -draw- commands.

The PLATO terminal has 126 built-in characters (including those used so far) and storage for 126 additional characters which can be different in every lesson. For example, Russian lessons fill this additional character storage space with the Cyrillic alphabet, whereas there is a genetics lesson which fills the storage area with fruitfly parts which permit displaying flies by writing appropriate characters at appropriate positions on the screen. We will learn how to access all 252 characters--those built-in and those which can be varied.

The 126 built-in characters include many useful symbols which do not appear on the keyset, because there aren't enough keys! This is due to the fact that the keys on the right of the keyset are reserved for various important functions (ERASE, BACK, STOP, etc.). In order to access the "hidden" characters it is necessary to first strike the ACCESS key (presently the shift-□ key) and then to strike a second key. Like SUPER and SUB, the ACCESS key is not held down but struck. You can press ACCESS, then "a" to get a Greek alpha; ACCESS-"b" for beta; ACCESS-"m" for mu; ACCESS-"=" for "≠"; ACCESS-"<" or ">" for "≤" or "≥". At a terminal it is useful to try ACCESS followed by every key (or shifted key) to find about 36 useful hidden characters. Luckily in most cases there is a mnemonic connection between the key which follows the ACCESS key and the hidden character which results, such as ≠ being ACCESS-"=". ACCESS followed by comma gives the symbol ‡ mentioned in the discussion of the -writec- command in chapter VI. ACCESS-Ø and ACCESS-1 give the symbols ◁ and ▷ used for embedding -show- commands in -write- statements.

You can get at the "alternate font" of 126 additional, modifiable characters by pressing the FONT key (the shifted MICRO key); then typing regular keys, which will produce characters from the alternate font. What characters appear depends on what character set has been previously loaded into the terminal. The FONT key toggles you between the standard built-in font and the alternate font: you stay in the alternate font until you strike FONT to return to the standard font. It is therefore not necessary to strike FONT for each symbol (unlike the way ACCESS works).

Here is an example of the use of a special character set:

```
at       912
write    Now LOADING CHARACTER SET.
         Please be patient - loading
         takes about 17 seconds.
charset  standard,russian
erase           $$ full-screen erase to remove message
unit     intro
at       9ø5
write    The Russian word карандаш means pencil.
```

The -charset- statement sends to the terminal the character set specified in the tag (character set "standard,russian" in this case). Character patterns are transmitted to the terminal at a rate of 7.5 character patterns per second, so a full 126-character set will take about 17 seconds to send. Precede the -charset- command with a -write- statement to explain this delay to the student; otherwise he will think something is broken! The full-screen -erase- will remove the message upon completion of the loading process. Once the character patterns have been loaded into the terminal it is possible to write Russian text on the student's screen at the same high speed as English, 18ø character per second, which corresponds to a reading speed of almost two thousand words per minute.

TUTOR keeps track of which character set has been loaded into the terminal and skips a -charset- statement if loading is not required. In the above example TUTOR would rush right through the message, skipping the -charset-, and erasing the screen. There would not be the 17-second delay which occurs if the Cyrillic characters have not been loaded.

The -write- statement in unit "intro" is created by:
1. typing "write The Russian word ";
2. striking the FONT key to select the alternate font;
3. typing the keys k, a, r, a, n, d, a, s (which causes карандаш to appear);
4. striking the FONT key to toggle back to the standard font;
5. typing " means pencil."

Each character in the alternate font is associated with a key on the keyset. For example, the creators of the "russian" character set chose to associate the Cyrillic "д" with the "d" key because of the phonetic similarity of these two letters. Similarly, the Cyrillic "p" and "н" sound like the "r" and "n" letters with whose keys they are associated. Just as accessing some of the

126 built-in characters requires the ACCESS key, so a full 126-character alternate font will also necessitate the use of the ACCESS key to reach some of the characters.

If the student is to respond at an -arrow- with a Russian response, he must hit the FONT key in order to do so. Usually it is preferable to precede the first judging command with the statement "force font" which essentially hits the FONT key for the student. He merely uses the regular typing keys, but his typing appears in the alternate font.

The "initial entry unit"--ieu

You may have noticed that the first few statements of the previous example (which write a message, load a character set, and then erase the screen) are not preceded by a -unit- statement. This is intentional. TUTOR statements which precede the first -unit- statement ("unit intro" in this case) constitute an "initial entry unit" which is performed whenever a student enters the lesson. The "initial entry unit" ("ieu" for short) is the logical place to put various kinds of initializations, such as a -charset- statement to load characters which will be used throughout the lesson. Although -define-, -vocabs-, and -list- statements are not actually executed (they are only instructions to TUTOR on how to interpret -calc-, -concept-, and -answer- statements in preparing a lesson for student use) they can be placed in the "ieu"; they do belong at the beginning of the lesson for the sake of readability.

The importance of the "ieu" lies in the fact that it is performed no matter where the student starts in the lesson--even if he does not start at the first unit statement. A student who leaves without finishing a lesson will restart the next day where he left off because TUTOR keeps track of where he was when he left. It is important in restarting to load the appropriate character set, which would not be accomplished if the -charset- statement came after the first -unit- statement since the student will not go through this first unit in restarting where he left off.

Suppose the student is to restart in unit "middle", which looks like this:

```
unit    middle
next    mid2
```

The way in which the "ieu" is utilized is that TUTOR acts as though the ieu were done at the beginning of the restart unit:

```
unit    middle
(do     "ieu")
next    mid2
```

This pseudo-do is the reason for following the -charset- statement with a full-screen erase. We don't want the "loading" message to mess up the display created by unit "middle".

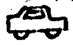
Smooth animations using special characters

The -charset- command is not limited to its use with foreign alphabets. Special characters are often used to create pictures:

```
at      1319
write   This
```

```
        uses special characters!
```

The car is composed of several adjacent characters. Because characters can be drawn very fast (18Ø per second) dramatic animations are possible:

```
mode    rewrite
do      drive,x⇐1ØØ,4ØØ
*
unit    drive
at      x,2ØØ
write
```


The car advances one dot at a time. If the car characters are designed in such a way as to leave a vertical column of blank dots at the back of the car, the "rewrite" mode will insure that the advancing car simultaneously erases its old position. If two columns are left blank the car could be advanced two dots at a time and still completely wipe out the previous car display. This type of animation can run as fast as twenty or thirty moves per second, which creates the illusion of a smoothly moving object.

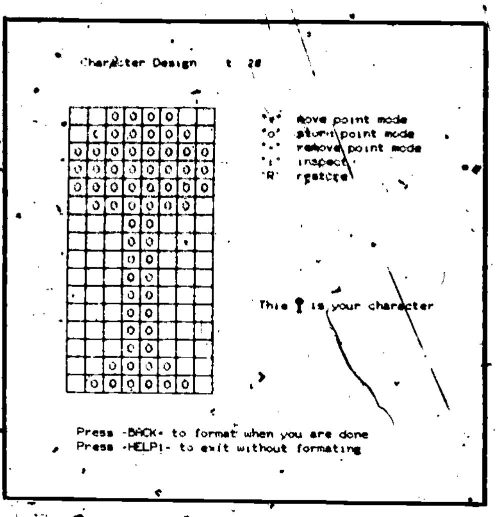
For the built-in characters there is an expandable and rotatable but slow line-drawn form available through the use of -size- and -rotate-, but these commands have no effect on text written in the alternate font. If a larger or rotated car is needed it must either be constructed with -draw- and -circle- commands or built up out of additional special characters.

Creating a new character set

Figure 1 demonstrates how a special character is designed at a PLATO terminal. The author moves the cursor on an 8 × 16 grid to specify which dots are to be lit. He can inspect "in the small" the appearance of the character he designs "in the large". The letter shown at the top of the page is the key with which this character will be associated when typing in the alternate font, just as character "Д" is associated with key "d" in "charset russian". The character pattern is stored in such a way that the author can at any later time recall the pattern and modify it. A character set can contain up to 126 special characters or as few as one or two characters.

Figure 2 shows how an author can create several 8 × 16 characters at once to be used together or separately. This option is particularly helpful when designing character-mode pictures.

Your own character set will be stored in an electronic storage area assigned to you. Such storage areas are called "lesson spaces" because they mainly hold TUTOR statements describing a lesson to be administered to students by PLATO. Your lesson space might be called "italian3" and it is by this name that you refer to the lesson space when you want to look



```
Character Design     t  28

"*"  move point mode
"o"  store point mode
"-"  remove point mode
"i"  inspect
"R"  restore

This ? is your character

Press -BACK- to format when you are done
Press -HELP1- to exit without formating
```

Fig. 1

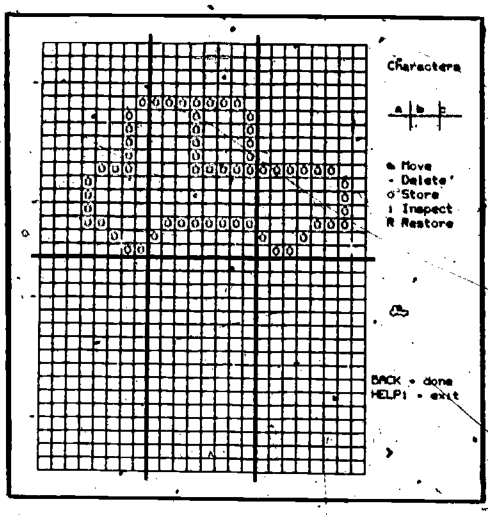
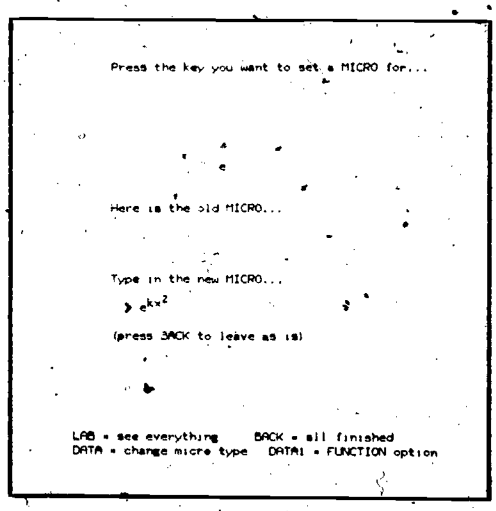


Fig. 2



```
Press the key you want to set a MICRO for...

Here is the old MICRO...

Type in the new MICRO...
  > e^kx^2

(press BACK to leave as is)

LAB = see everything     BACK = all finished
DATA = change micro type   DATA1 = FUNCTION option
```

Fig. 3

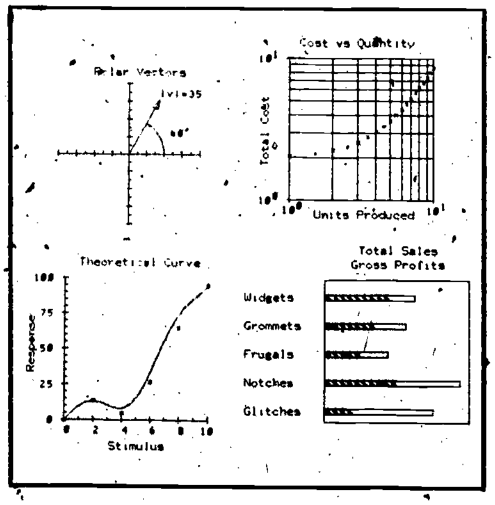


Fig. 4

at the TUTOR statements or change them. Within this lesson space you can also have one or more character sets, which you will have named. Suppose in lesson space "italian3" you have stored a character set named "rome". Then the TUTOR statement used to transmit this character set to a terminal is

$$\underset{\text{lesson space}}{\texttt{charset italian3}},\underset{\text{character set}}{\texttt{rome}}$$

Micro tables

It is sometimes desirable to associate a string of several characters with a single key. For example, the symbol $\bar{\bar{v}}$ may be produced by v, backspace, superscript, minus sign. It is possible to set up a "micro table" so that $\bar{\bar{v}}$ may be produced simply by hitting the MICRO key followed by hitting "v". Similarly, the micro table might specify that MICRO-"e" should be equivalent to typing e, shift-SUPER, k, x, SUPER, 2, shift-SUB to make e^{kx^2}. The micro table makes possible a kind of shorthand which can be useful both to authors composing -write- statements and to students typing complicated responses.

Like character sets, micro tables reside in lesson spaces. If lesson space "italian3" contains a micro table named "dante", these micros can be made available to students by the statement

```
micro   italian3,dante
```

As with -charset-, the -micro- statement should be placed in the "ieu" (initial entry unit).

Figure 3 shows how an author defines an item in a micro table, by associating a string of characters with a particular key. Later the effect of striking MICRO followed by this key is identical to typing this string of characters.*

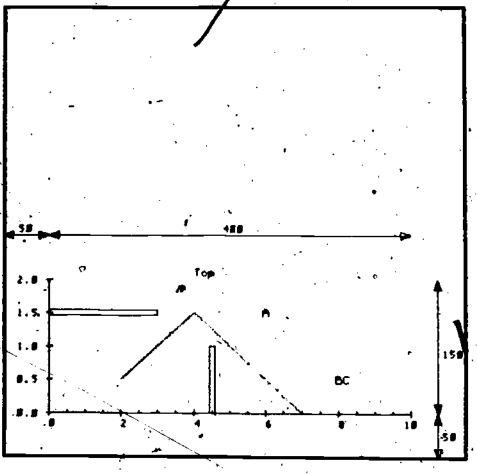
If you do not specify your own micro table a standard one is provided that lets you use the MICRO key as though it were the ACCESS key: for example, MICRO-"p" gives ACCESS-"p", which is π. This means you can and should mention only the MICRO key to students in your typing directions to them. It is not necessary to mention ACCESS.

The graphing commands: plotting graphs with scaling and labeling

You may often want to plot a horizontal or vertical bar graph or other kind of graph to display relationships. There exists a group of TUTOR commands which collectively make it very easy to produce such displays. In particular, scaling of your variables to screen coordinates is automatic, as is the numerical labeling of the axes, with tick marks along the axes. Figure 4 shows an example.

* With a "force micro" in effect, the student does not have to press MICRO. This makes it easy to redefine the keyboard.

Suppose you want a graph to occupy the lower half of the screen. The horizontal x-axis should run from zero to ten and the vertical y-axis from zero to two. Both axes should be labeled appropriately. These statements will make the display shown:

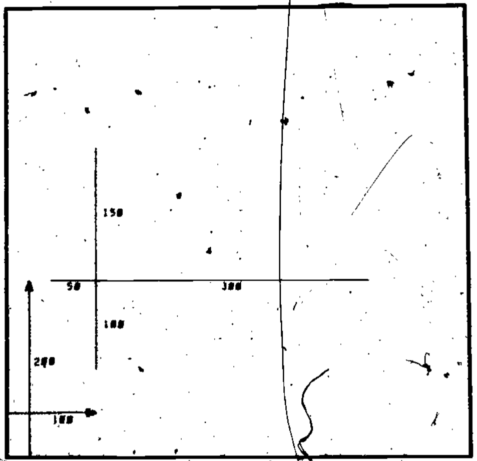


```
unit     setup
origin   5Ø,5Ø               $$ x,y origin
axes     4ØØ,15Ø             $$ lengths in dots
scalex   1Ø                  $$ maximum x
scaley   2                   $$ maximum y
labelx   2,.5                $$ major mark every 2,
*                            minor every .5
labely   .5                  $$ major mark every .5
graph    6,1.5,A             $$ x=6, y=1.5
graph    8,.5,BC             $$ x=8, y=.5
hbar     3,1.5               $$horizontal bar to
*                            3,1.5
vbar     4.5,1               $$ vertical bar to
*                            4.5,1
gdraw    2,.5;4,1.5;7,Ø
locate   4,2
write    Top
```

After specifying -origin- and -axes- in terms of fine-grid screen coordinates the -scalex- and -scaley- commands associate scale values with the end points of the axes. These scale values determine how (x,y) coordinate positions given in later statements will be scaled to screen coordinates. The -labelx- and -labely- commands cause numerical labels and tick marks to appear. The statement "graph 6,1.5,A" plots an A at x=6, y=1.5 in scaled coordinates. The -hbar- and -vbar- commands draw horizontal and vertical bars to the specified scaled points. The -gdraw- command is like -draw-, except points are specified in terms of scaled quantities. The -locate- command is like -at- but uses scaled quantities.

Reread the example and try to identify in the picture what part of the display results from each statement. Of course each number in the tags of these statements could have been a complicated mathematical expression.

The -markx- and -marky- commands are similar to -labelx- and -labely- but merely display tick marks without writing numerical labels. The -axes- command has an alternative form to allow for axes in the negative directions:



```
origin  1ØØ,2ØØ
axes    -5Ø,-1ØØ,3ØØ,15Ø
```

$\underbrace{-5Ø,-1ØØ}_{\text{minimum x,y from origin}},\underbrace{3ØØ,15Ø}_{\text{maximum x,y from origin}}$

Although the commands were originally designed to make it easy to draw graphs, the automatic scaling features make these commands useful in many situations. Note in particular that you can move complicated displays around on the screen merely by changing the -origin- statement.

Additional graphing commands include -vector- for drawing a line with an arrowhead at one end; -polar- for polar coordinates; -lscalex- and -lscaley- for logarithmic scales; and -funct- and -delta- for plotting functions. The -bounds- command has the same effect as -axes- in establishing lengths, but no axes are drawn on the screen. The -frame- command is used to draw rectangular boxes easily.

<u>Summary of line-drawing commands: -draw-, -gdraw-, -rdraw-</u>

Recall that the -draw- statement has the form

```
        draw    point1;point2;point3;etc.
```

where each point may be coarse-grid (such as "1215") or fine-grid (such as "135,245"). Each point specification is set off by a semicolon in order to avoid ambiguities when mixing coarse-grid and fine-grid points, as in "draw 1525;1932;35,120;1525" (the first two points are given in coarse-grid; the third, in fine-grid; and the last point in coarse-grid coordinates).

A discontinuous line drawing can be made with a single -draw- statement by using the word "skip":

```
        draw    1518;1538;skip;1738;1718
```

Using "skip" in a -draw- statement means "skip to the next point without drawing a line." This example is essentially equivalent to

```
        draw    1518;1538
        draw    1738;1718
```

The only difference between these otherwise equivalent forms is related to the fact that the system variables "where", "wherex", and "wherey" are not brought up to date until the <u>completion</u> of the -draw- statement. The sequence

```
        at      1319    $$ affects "where"
        draw    1518;1538;skip;1738;where
```

is equivalent to

```
        at      1319
        draw    1518;1538
        draw    1738;1319
```

since during the -draw- statement "where" has the value 1319. On the other hand, the sequence

```
at      1319
draw    1518;1538
draw    1738;where
```

is equivalent to

```
at      1319
draw    1518;1538
draw    1738;1538
```

since upon completion of the first -draw- statement, the value of "where" is 1538. This difference between a single -draw- using "skip" and separate -draw- statements is sometimes useful in drawing figures relative to some point.

As mentioned earlier, starting with a semicolon implies a continued drawing from the present screen location:

```
at      1319
draw    ;1542;1942
```

is equivalent to

```
at      1319
draw    where;1542;1942
```

or "draw   1319;1542;1942".

Sometimes you have more points for a -draw- than will fit on one line. A "continued" -draw- can be written, with the command blank on succeeding lines:

```
draw    1512;1542;skip;100,200;
        400,200;400,400;
        100,400;100,200
```

This will behave as though all the points had been listed on one line.

To summarize, the -draw- statement contains fine-grid or coarse-grid points separated by semicolons; "skip" can be used for a discontinuous drawing; "where" and the fine-grid "wherex" and "wherey" are brought up to date upon completion of the -draw-; and starting the tag with a semicolon has the special meaning of continuing a drawing from the present screen position.

The -gdraw- command is like the -draw- command except that points are relative to the graphing coordinate system established by -origin-, -axes- (or -bounds-), -scalex-, and -scaley- (or logarithmic scales set up by -lscalex- and -lscaley-). Of particular value are the "skip" option and starting with a semicolon for continuing a drawing. The use of "where", "wherex", and "wherey" in a -gdraw- statement is normally not meaningful, since these system variables refer to the absolute screen coordinate system, not the graphing system. In the graphing coordinate system, there are only fine-grid, not coarse-grid points, so all points have the form "x,y".

It is possible to use -draw- to draw something relative to the present screen position:

```
        draw    wherex+25,wherey-75;wherex+2ØØ,wherey+15Ø
```

(Remember that "wherex" and "wherey" do not change until the completion of the -draw- statement.) There is an -rdraw- command ("r" for "relative") which makes such drawings simpler. The example just shown can be written

```
        rdraw   25,-75;2ØØ,15Ø
```

Each point is taken to be relative to the present screen position, which is referred to in an -rdraw- as the point Ø,Ø (wherex+Ø,wherey+Ø). Each -rdraw- statement skips back to Ø,Ø so that another -rdraw- can be performed relative to the original screen position. It is as though you had written

```
        rdraw   25,-75;2ØØ,15Ø;skip;Ø,Ø
```

to get back to the relative origin. Note that this is different from the behavior of the -draw- command, which leaves you at the last point drawn to.

The -rdraw- command is particularly useful for applications such as writing the same Chinese characters at different places on the screen. For each character, make a subroutine involving one or more -rdraw- statements. The characters can be positioned with -at- statements:

```
        at      4ØØ,4ØØ
        do      chin1
        at      4ØØ,3ØØ
        do      chin2
            etc.
```

Or you might include the -at- statement in the character subroutines:

```
        do      chin1(4ØØ,4ØØ)
        do      chin2(4ØØ,3ØØ)
```

In this case each subroutine has a form like this:

```
        unit    chin1(a,b)
        at      a,b
        rdraw   -75,3∅;75,3∅;etc.
```

Unlike -draw-, the -rdraw- command is affected by preceding -size- and -rotate- commands. Your Chinese characters can be enlarged and rotated:

```
        size    3.5         $$ 3.5 time normal size
        rotate  45          $$ 45 degrees
        do      chin1(4∅∅,4∅∅)
        do      chin2(4∅∅,3∅∅)
```

Figure 5 shows a design created with the following commands:

```
        do      figure,a⇐∅,36∅,15
        *
        unit    figure
        rotate  a
        rdraw   -5∅,∅;5∅,∅;∅,2∅∅;-5∅,∅
```

The -rotate- command affects -rdraw- even with "size ∅", even though -write- is not rotated in size ∅. This is done to facilitate normal text operations. As far as -rdraw- is concerned, size ∅ is equivalent to size 1. As far as -write- is concerned, size ∅ means "write text at 18∅ characters per second, unrotated", whereas size 1 means "write line-drawn text at 6 characters per second, rotated".

Note that -rdraw- and -size- are essentially reciprocal to -gdraw- and -scalex-. In the case of -rdraw- a drawing gets bigger when -size- specifies a larger size. But specifying a larger number in a -scalex- command implies that the same number of screen dots (given by -axes-) will now correspond to larger (scaled) numbers in a -gdraw-. This means that a larger -scalex- implies a smaller -gdraw- figure. Note that -origin- affects -gdraw- the same way that -at- affects -rdraw-.

While -scalex- and -scaley- permit you to scale -gdraw- differently in the horizontal and vertical directions, it is not yet possible to specify both an x-size and a y-size for -rdraw-. On the other hand, -rdraw- can be rotated, but there is not yet a -grotate- command for rotating a -gdraw-. It is possible that these missing features will eventually be added to TUTOR. Similarly, there may become available an -rcircle- affected by -size- and -rotate- and a -gcircle- affected by -scalex- and -scaley-.

The -window- command

Sometimes it is useful to specify a "window" through which drawings are viewed. Parts of a figure extending outside the window are not drawn. A rectangular window is specified by giving the lower left and upper right corners of the desired window:

window $\underbrace{1ØØ,2ØØ}_{\text{lower left}},\underbrace{4ØØ,3ØØ}_{\text{upper right}}$

The corners could also be given in coarse-grid coordinates, as in "window 1524,1248".

Drawings constructed from the various -draw- commands and -circle- commands are affected by a preceding -window- command. Line-drawn text (size non-zero) produced by -write-, -writec-, -show-, etc., will also be windowed. Like -size- and -rotate-, windowing is not reset upon entering a new main unit. BE SURE to use a blank -window- command (blank tag) to turn off windowing operations. A very common error is to forget to turn off windowing and then wonder why some of the drawings aren't showing up! The correct structure is

```
window  lower left corner,upper right corner
.
.
.
windowed display statements
.
.
.
window                 $$ blank tag to turn off
```

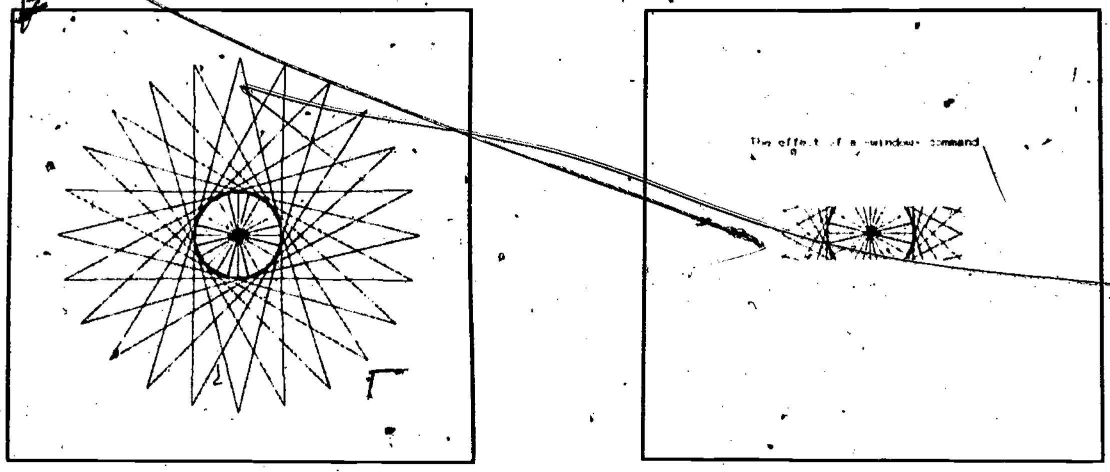


Fig. 5

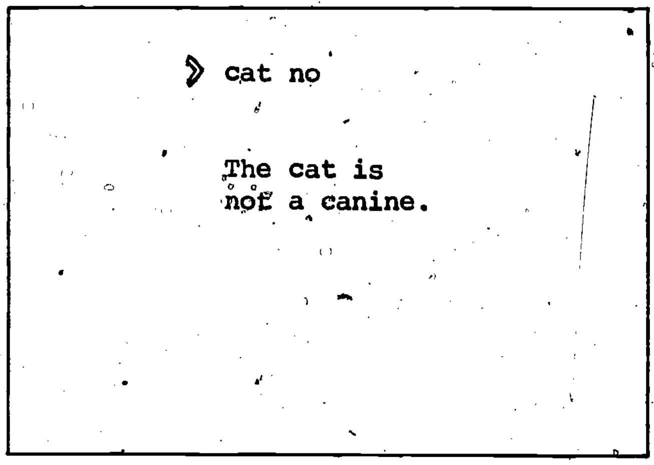
More on erasing: the -eraseu- command

When a student's response is judged "no" or "wrong", he can correct his response by hitting ERASE or ERASE1 to erase a letter or word, or by hitting NEXT, EDIT, or EDIT1 to erase the entire response. If additional judging keys have been defined with a -jkey- command, these will act like NEXT and erase the response. If there is only one -arrow- command and no -endarrow-, these options are available even after an "ok" judgment, except that a NEXT key or other judging-key takes the student to the next main unit rather than merely erasing the response.

If the student erases part or all of his response, the "ok" or "no" is erased. Moreover, the last response-contingent message to the student is erased, since it is no longer relevant. For example:

```
        .
        .
        .
        wrong   cat
        write   The cat is
                not a canine.
        .
        .
        .
```

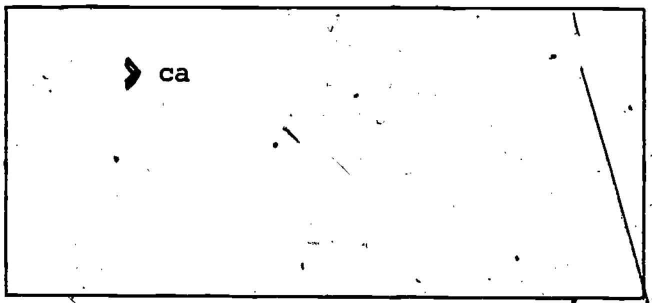
The student types "cat" and presses NEXT:



```
        > cat no

          The cat is
          not a canine.
```

Notice that there is a default -at- three lines below the response. Suppose the student now presses ERASE:



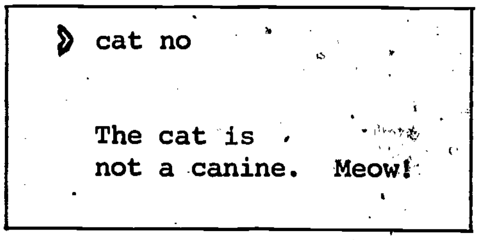
```
> ca
```

The "t", the "no", and the text of the -write- statement have all disappeared automatically. This is appropriate since the comment "The cat is not a canine" is no longer needed.

It is helpful to know that the method TUTOR uses for automatically erasing such text is by re-executing the last -write-, -writec-, or -show- statement in the erase mode. Suppose we change the lesson slightly:

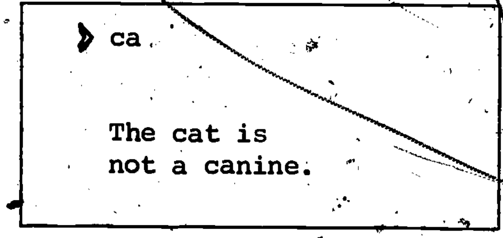
```
   .
   .
   .
wrong   cat
write   The cat is
        not a canine.
write     Meow!
   .
   .
   .
```

Now the sequence looks like this:



```
> cat no


  The cat is
  not a canine.   Meow!
```



```
> ca


  The cat is
  not a canine.
```

Only the last -write- statement is removed, leaving "The cat is not a canine" on the screen. Notice that the normal automatic erasing can be prevented simply by adding an extra -write- statement. Even a blank -write- statement will do.

As another example, consider this:

```
        .
        .
        .
        wrongv  4
        write   Number of apples=
        show    apnum
        .
        .
        .
```

Only the -show- will be erased, leaving "Number of apples=" on the screen. If this is not desirable, use an embedded -show-:

```
        .
        .
        .
        wrongv  4
        write   Number of apples= ⊲s,apnum⊳
```

Now the last -write- statement includes the showing of the number, and all the writing will be erased. It is important not to change "apnum" after the -write-. If you change its value from what it was when shown by the -write-, the re-execution in mode erase will turn off the wrong dots in the numerical part of the writing. Here is the type of sequence to be avoided:

```
        .
        .
        .
        wrongv  4
        write   Number of apples= ⊲s,apnum⊳
        calc    apnum⇐apnum+25
        .
        .
        .
```

The number will not be erased properly due to the change in "apnum". Similar problems can arise with the other -show- commands, including -showa-.

Sometimes the automatic erasing of the last text statement is insufficient. For example, if the reply to the student included a drawing produced with -draw-, or if there were several -write- statements, you need some additional mechanism to remove the reply when the student presses ERASE. There is an

-eraseu- command which you can use to specify a subroutine to be done when the student changes his response:

```
         .
         .
         .
         eraseu  eblock
         arrow   1215
         .
         .
         .
         unit    eblock
         at      1512
         erase   35,4
         at      318
         erase   42
         .
         .
         .
```

Unit "eblock" will be done whenever the student changes his response. Only the first press of the ERASE key triggers the erase unit, since additional executions of the unit would be erasing nothing.

Another example involves an erase unit specific to a particular response:

```
         .
         .
         .
         wrong   3 dogs
         do      woof
         eraseu  remove
         .
         .
         .
         unit    remove
         mode    erase
         do      woof
         mode    write
         eraseu
         .
         .
         .
```

The statement "eraseu remove" defines unit "remove" as the unit to be done when the student presses ERASE (or NEXT, etc.). Unit "remove" in the example shown simply re-does unit "woof" in the erase mode, thus taking off the screen everything originally displayed by unit "woof". The final blank -eraseu- clears the pointer so there is no longer an erase unit specified.

Notice the similarities between the -imain- and -eraseu- commands. Both specify units to be done under specific conditions.

Keeping things on the screen: "inhibit erase"

Let us consider a modified version of the simple language drill discussed in chapter VII:

```
unit    espo
next    espo
back    satisfy
at      512
write   Here is a simple drill
        on the first five
        Esperanto numbers.
        Press BACK when you
        feel satisfied with your
        understanding.
at      1812
write   Give the Esperanto for
randu   item,5
at      2015
writec  item-2,one,two,three,four,five
arrow   2113
join    item-2,unu,du,tri,kvar,kvin
*
unit    unu
answer  unu
*
unit    du
answer  du
*
unit    tri
answer  tri
*
unit    kvar
answer  kvar
*
unit    kvin
answer  kvin
```

This version will greatly annoy the student after the first couple questions. The difficulty is that each time he gets an "ok" and presses NEXT to move on to the next unit, the screen is erased and he suffers through the introductory paragraph being written again on the screen. It turns out to be very annoying to see text replotted this way because you already know what it says.

This is a situation where most of the material on the screen is not changing and should not be replotted: only the item and the student's typing need be erased to make room for a new item ~~and a new~~ response. One way to do this involves judging correct responses "wrong", as was done in the dialog using -concept- discussed in chapter VII. You should use "specs nookno" to prevent the "no" from appearing. Or you can use the regular -okword- and -noword- commands to change the standard TUTOR "ok" and "no". For example, use the statement "noword Fine!" to cause "Fine!" to appear for a correct response. You would need to do a "noword no" whenever the student answers incorrectly. With all responses judged "wrong" we stay at the -arrow- and do not move on to another main unit.

Another way to manage a screen on which little is changing involves "inhibit erase". This statement prevents the normal full-screen erase upon leaving the present main unit. The next main unit must also execute an "inhibit erase" if no erase is to be performed upon leaving the second unit. We can rewrite our drill using this feature:

```
unit    preespo
at      512
write   Here is a simple drill
        on the first five
        Esperanto numbers.
        Press BACK when you
        feel satisfied with your
        understanding.
at      1812
write   Give the Esperanto for
goto    espo1
*
unit    espo
at      2ø15
erase   5              $$ item area
at      2115
erase   15             $$ response area
entry   espo1
inhibit erase          $$ leave instructions on screen
next    espo
back    satisfy
randu   item,5
at      2ø15
writec  item-2,one,two,three,four,five
arrow   2113
join    item-2,unu,du,tri,kvar,kvin
```

In unit "preespo" we display the instructions about the drill. We then go to "espo1", where we inhibit erase and display the first item. Upon getting an "ok" the student moves on to the next main unit, "espo". The screen is not erased since there was an "inhibit erase". In unit "espo" we erase the area containing the displayed item, and we also erase the response area of the screen. We fall through the -entry- command and display a new item. This process repeats continually, and only those parts of the screen which must be changed are erased.

It is important to place an explicit blank -erase- statement ("erase ") at the beginning of unit "satisfy". Since we have inhibited the normal full-screen erase, no erase will occur automatically when the student presses BACK to leave the drill. If unit "satisfy" does not explicitly erase the screen, the student will see a superposition of the drill display and the display produced by unit "satisfy".

Similarly, if we specify a help unit, that unit should start with a full-screen erase. Upon completion of the help sequence, we should come back to unit "preespo" rather than "espo" in order to restore the screen display properly. Do this:

```
        .
        .
        .
        entry   espo1
        base    preespo       $$ to come back to preespo from help
        help    esphelp
        .
        .
        .
```

The -base- command puts us in a help sequence, with the base unit being "preespo". When a base unit has already been specified, pressing HELP doesn't change the base unit (in other words, there is only one "level" of help). When we reach an -end- command or press BACK, we will return to the base unit, which is preespo. Note that unit "satisfy" should have a blank base statement to insure that we are in a non-help sequence. Otherwise, pressing BACK in unit "satisfy" will bring us to the base unit "preespo" again.

Interaction of "inhibit erase" with -restart-

There is a -restart- command which is used to specify in which unit a student should resume study upon returning to a PLATO terminal. For example, suppose the last -restart- statement encountered on Monday by student "Ann North" in course "lingvo" was "restart espo" in lesson "espnum". On Wednesday she returns to a PLATO terminal and identifies herself by name (Ann North) and course (lingvo). Her registration records will show that she is to be restarted in unit "espo" of lesson "espnum" and she will automatically be taken to that point. As discussed previously, the ieu (initial entry unit) will be done, which among other things permits character set loading.

Unfortunately, restarting at unit "espo" means that the basic drill instructions contained in unit "preespo" will not appear (see last example).

This is basically an initialization problem. You should use -restart- commands in such a way as to restart students only at the beginning of a section of this kind. In this particular case, we should have had a "restart preespo" rather than "restart espo". This is analogous to our use of "base   preespo" for returning from a help sequence. (The more common form of the -restart- is the blank -restart-, which means "restart in the present main unit". We would place a blank -restart- in unit "preespo".)

Aside from initialization questions related to TUTOR and the display screen, it should be pointed out that the student has comparable initialization problems. Since the student may be away for several days, it is usually advisable to have your restart points only at the beginning of sections of the lesson. This way the student can ease back into the context, whereas restarting in the middle of a discussion may be quite confusing. In lessons whose structure includes an index, the index unit may be the best restart point.

When a student restarts in a lesson, he starts at the unit specified by the last -restart- command. However, his saved variables, v1 through v15∅, have whatever values were current at the time he left the last PLATO class session. Therefore, some care is required to initialize appropriate variables in the restart unit.

The -char- and -plot- commands

Usually special characters are handled with a -charset- command and displayed with a -write- statement using the FONT key. Alternatively, -char- commands can be used to transmit character patterns to the terminal. If a -char- command sends a pattern to character slot 35 of the terminal, that character can be displayed using the -plot- command: "plot   35". The arguments of the -char- command can be computed expressions so that a character can be constructed algorithmically. Similarly, the -plot- command may have a mathematical expression for its tag in order to choose the Nth character. See appendix A for sources of detailed information on the -char- command.

The -dot- command

The statement "dot   125,375" will plot a single dot at the specified location ("dot   1817" uses coarse grid). A sequence of -dot- commands can produce sixty dots per second on the plasma display panel. A -draw- with one point ("draw   125,375" or "draw   1817") makes a single dot by drawing a line from this point to this point and for technical reasons will produce only twenty dots per second.

IX. Additional calculation topics

Before discussing additional TUTOR calculational capabilities, let's review briefly those aspects which have been covered so far:

1) Expressions follow the rules of high school algebra. Multiplication takes precedence over division, which takes precedence over addition and subtraction. Superscripts may be used to raise numbers to powers. The symbol π may be used to mean 3.14159..... The degree sign (°) may be used to convert between degrees and radians.

2) There are 15Ø student variables, v1 through v15Ø, which may be named with the -define- command. These variables can be set or altered by assignment (⇐) and by -store-, -storen-, or -storea- commands. If a "define student" set of definitions is provided, the student may use variable names in his responses.

3) Logical expressions are composed using the operators =, ≠, >, <, ≥, ≤, and, or, and the "not" function. Logical expressions have the value true (-1) or false (Ø).

4) There are available various system variables such as "where", "wherey", "anscnt", "jcount", "spell", etc. Available system functions include sin(x), sqrt(x), etc. A full list of system variables and functions is given in Appendix C.

5) The -show- command (and its relatives -showt-, -showz-, -showe-, and -showo-) will display the numerical value of an expression. The -showa- command will display stored alphanumeric information. These commands may be embedded within -write- and -writec- statements.

6) The -calcc- and -calcs- commands make it easy to perform conditionally one of a list of calculations or assignments.

7) The -randu- command with one argument picks a fraction between Ø and 1; with two arguments it picks an integer between 1 and the limit specified. There is a set of commands associated with permutations: -setperm-, -randp-, -remove-, and -modperm-.

8) The iterative form of the -do- command facilitates repetitive operations.

Now let us turn to additional TUTOR calculational capabilities.

Defining your own functions

While many important functions such as ln(x) and log(x) are built-in to the TUTOR language, it is frequently convenient to define your own functions. To take a simple example, suppose you define a cotangent function:

```
        define  cotan(a)=cos(a)/sin(a)
```

Then later in your lesson you can write

```
        calc    r⇐cotan(3x+y-5)
```

and TUTOR will treat this as though you had written

```
        calc    r⇐[cos(3x+y-5)/sin(3x+y-5)]
```

Such use of functions not only saves typing but improves readability.

CAUTION: In defining a function, the arguments must not be already defined. For example, the following definition will be rejected by TUTOR (with a suitable error message):

define x=v1
        cube(x)=x^3

This must be rewritten as

define x=v1
        cube(dummy)=dummy^3

or anything similar. A function definition may involve previously defined quantities on the right side of the "=" sign, however:

define x=v1
        new(c)=c^4+2x

In this case you might have a -calc- that looks like

```
        calc    x⇐15.7
                y⇐3new(8)
```

and this would be equivalent to

calc   x⇐15.7
        y⇐3[$(8)^4$+2x]

Sometimes it is convenient to define "functions" that have no arguments:

define r=v1
        quad=r^2-1ØØ
        r3=$r^{1/3}$
        root=sqrt(r)
        prod=r3×root
        trans=(r⇐prod)

Note that "prod" depends on two previous definitions, each of which in turn depend on the definition of "r". There is no limit on how deep you can go in definition levels. The unusual definition of "trans" permits you to write an unusual -calc-:

```
        calc   trans
```

where the assignment is implicit in the definition of "trans".

Essentially anything is a legal definition. The only rule is that the definition make sense when enclosed in parentheses, since a defined name when encountered in an expression is replaced by its meaning and surrounded by parentheses. This means that you cannot define "minus=-" because (-), a minus sign enclosed in parentheses, is not permitted in an expression. "minus=-1" is all right because (-1) is meaningful.

A function may have up to six arguments. Here is a function of two arguments:

```
        define  modulo(N,base)=N-[base×int(N/base)]
```

will mean that modulo (17,5) in an expression will have the value 2: the "int" or "integral part" function throws away the fractional part of 17/5, leaving 3, so that we have (17-5×3)=(17-15)=2. This modulo function therefore gives you what is left over in division of "N" by "base".

Here are a couple of other examples of multi-argument function definitions:

```
        define  big(a,b)=-[a×(a≥b)+b×(b>a)]
                small(a,b)=-[a×(a≤b)+b×(b<a)]
```

The minus sign is there because logical true is represented by -1. If you have "big(x+y,z)" in an expression, with (x+y)=7 and z=3, this expands to

```
      -[7×(7≥3)+3×(3>7)]
```

which reduces to -[7×(-1)+3×(∅)] which is 7. So our "big" function picks out the larger of two arguments. CAUTION: The value of "big(2.99999999999999,3)" is 6 (!) instead of 3. The equality test in TUTOR, including the equality part of (≤) or (≥), does a small amount of rounding to compensate for roundoff errors inherent to computers, so that "2.99999999999999≥3" is <u>true</u> (-1). The tests (<) and (>) do not do this, so "3>2.99999999999999" is also true (-1). Therefore, a better definition to pick out the larger of two numbers is "define big(a,b)=a-(b>a)(b-a)". Similarly, write "small(a,b)=a-(b<a)(b-a)".

<u>Arrays</u>

It is often important to be able to deal with arrays of data such as, a list of exam scores, the number of Americans in each 5-year age group together with their corresponding mortality and fertility rates, a list of which pieces are where on a chess board, or the present positions of each of several molecules in the simulation of the motion of a gas.

Suppose we have somehow entered the exam scores for twenty students into variables v31, v32, v33, ... up to v5∅. Here is a unit which will let you see the score of the 5th or 13th or Nth student:

```
unit     seq
back     index
at       1215
write    Which student number?
         (Press BACK when done.)
arrow    1518
store    N
wrongv   1∅.5,9.5    $$ range 1 to 2∅
write    The score of the ⊲s,N⊳ th student is ⊲s,v(3∅+N)⊳.
```

(The -wrongv- rather than -ansv- makes it easy to ask another question.) The new element here is the "indexed variable"

```
v(3∅+N)
```

which means "evaluate 3∅+N, round to the nearest integer, and choose the corresponding variable". For example, if N is 9, v(3∅+N) is v(39) or v39. If N is 13.7, v(3∅+N) means v44.

We might list and total all the scores:

```
     .
     .
calc     total⇐∅          $$ initialization step
do       showem,N⇐1,2∅
at       3∅35
write    The average score is ⊲s,total/2∅⊳ .
*
unit     showem
at       835+1∅∅N
show     v(3∅+N)
calc     total⇐total+v(3∅+N)
```

As usual it is far preferable to define a name for these data:

```
define   scores(i)=v(3∅+i)
```

in which case we would write our last unit as

```
unit     showem
at       835+1∅∅N
show     scores(N)
calc     total⇐total+scores(N)
```

Due to the special meaning attached to "v(expression)" you must exercise some care in using a variable named "v", in that you must write "v×(a+3b)" and not "v(a+3b)" if you mean multiplication. We will see later that the same restriction applies to the names "n", "vc", and "nc". This restriction does not apply to students entering algebraic responses, where "v(a+3b)" is taken to mean "v×(a+3b)". Students can use indexed variables only if they are named, as is "scores" in the above example; such definitions must, of course, be in the "define student" set.

Suppose you have three sets of exam scores for the twenty students. This might conveniently be thought of as a 3 by 2∅ ("two-dimensional") array. Suppose we put the first twenty scores in v31 through v5∅, the second set in v51 through v7∅, and the third set in v71 through v9∅. It might be convenient to redefine your array in the following manner:

```
define   scores(a,b)=v(1∅+2∅a+b)
```

Then if you want the 2nd test score for the 13th student you just refer to scores(2,13) which is equivalent to v(1∅+4∅+13) or v(63). If you wanted to display all the scores you might use "nested" -do- statements:

```
         .
         .
do       column,i⇐1,3
*
unit     column
do       rows,j⇐1,2∅
*
unit     rows
at       82∅+1∅i+1∅∅j
show     scores(i,j)
```

Unit "column" is done three times, and for each of these iterations unit "rows" is performed twenty times.

There is an alternative way to define our array:

```
define   i=v1,j=v2
         scores=v(1∅+2∅i+j)
```

Then our unit "rows" would look like

```
unit     rows
at       82∅+1∅i+1∅∅j
show     scores
```

The indices specifying which test for which student are implicit. This form is particularly useful when you have large subroutines where "i" and "j" are fixed and it would be tiresome to type over and over again "scores(i,j)". Just set "i" and "j", then -do- the subroutine.

It is frequently necessary to initialize an entire array to zero. One way to do this is with -do- statements:

```
unit    clear
do      clear2,i⇐1,3
*
unit    clear2
do      clear3,j⇐1,2ø
*
unit    clear3
calc    scores(i,j)⇐ø
```

A simpler way to accomplish the same task is to say

```
zero    scores(1,1),6ø
```

You simply give the starting location (the first of the 6ø variables) and the number of variables to be cleared to zero. As another example, you can clear all of your variables by saying

```
zero    v1,15ø
```

Not only is the -zero- command simpler to use, but TUTOR can carry out the operation several hundred times faster! TUTOR keeps a block of its own variables, each of which always contains zero. When you ask for 15ø variables to be cleared, TUTOR does a rapid block transfer of 15ø of its zeroed variables into your specified area. This ultra-high-speed block transfer capability can be used in other ways:

```
transfr v1ø;v85;25
```

performs a block transfer of the 25 variables starting with v1ø to the 25 variables starting with v85. In this way you can move an entire array from one place to another with one -transfr- command, and at speeds hundreds of times faster than are possible by other means.

Segmented variables

Storing three scores for each of your twenty students required the use of 6ø variables, out of 15ø available. We're running out of room! You can save space by defining "segmented" variables which make it easy to keep several numbers in each student variable. For example, you can write a definition of the form

```
define  segment,score=v31,7
```

This identifies "score" as an array which starts at v31 and consists of segments holding positive integers (whole numbers) smaller than 2^7 (which is 128). It turns out that each student variable will hold 8 such segments, so "score(8)" is the last segment in v31, while "score(9)" is the first segment in v32. Since "score(6∅)" is the fourth segment in v38, we need only eight variables to hold all sixty scores! You can use "score(expr)" in calculations: the expression "expr" will be rounded to the nearest integer and the appropriate segment referenced. As a simple example,

```
calc    score(23)⇐score(3)+5
```

will get the third segment, add 5 to it, and store the result in the twenty-third segment.

If we define a segmented one-dimensional array "score", we can define a two-dimensional array as before:

```
define segment,score=v31,7
       scores(a,b)=score(2∅a-2∅+b)
```

with these definitions, "scores(1,1)" means "score(2∅-2∅+1)" or "score(1)", which is the first segment in v31. As before, "scores" could use implicit indices:

```
define  i=v1,j=v2
        scores=score(2∅i-2∅+j)
```

in which case you use "scores" rather than "scores(expr1,expr2) in calculations. NOTE: At the present writing the commands -zero- and -transfr- cannot be used with segmented variables because these commands refer to entire variables. You could, however, zero all of the scores by saying "zero v31,8" which sets v31 through v38 to zero, which has the effect of zeroing all the segments contained in those eight variables. You can make such manipulations more readable by defining your segmented array this way:

```
define  start=v31
        segment,score=start,7
```

Then you can write "zero start,8" rather than "zero v31,8". Similar remarks apply to the -transfr- command.

It is possible to store integers (whole numbers) that can be negative as well as positive:

```
define  segment,temp=v5,7,signed
```

The addition of the word "signed" (or the abbreviation "s") permits you to hold in "temp(i)" any integer from -63 to +63. The range 2^7 (128) has been cut essentially in half to accomodate negative as well as positive values. The following table summarizes the unsigned and signed ranges of integers permissible for various segment size specifications up to 3∅ (sizes up to 59 are allowed, though beyound 3∅ there is only one segment per variable).

Segment size n	2^n	unsigned range	signed range	No. of segments per variable
1	2	Ø to 1	-	6Ø
2	4	Ø to 3	-1 to +1	3Ø
3	8	Ø to 7	-3 to +3	2Ø
4	16	Ø to 15	-7 to +7	15
5	32	Ø to 31	-15 to +15	12
6	64	Ø to 63	-31 to +31	1Ø
7	128	Ø to 127	-63 to +63	8
8	256	Ø to 255	-127 to +127	7
9	512	Ø to 511	-255 to +255	6
1Ø	1 Ø24	Ø to 1 Ø23	-511 to +511	6
11	2 Ø48	Ø to 2 Ø47	-1 Ø23 to +1 Ø23	5
12	4 Ø96	Ø to 4 Ø95	-2 Ø47 to +2 Ø47	5
13	8 192	Ø to 8 191	-4 Ø95 to +4 Ø95	4
14	16 384	Ø to 16 383	-8 191 to +8 191	4
15	32 768	Ø to 32 767	-16 383 to +16 383	4
16	65 536	Ø to 65 535	-32 767 to +32 767	3
17	131 Ø72	Ø to 131 Ø71	-65 535 to +65 535	3
18	262 144	Ø to 262 143	-131 Ø71 to +131 Ø71	3
19	524 288	Ø to 524 287	-262 143 to +262 143	3
2Ø	1 Ø48 576	Ø to 1 Ø48 575	-524 287 to +524 287	3
21	2 Ø97 152	Ø to 2 Ø97 151	-1 Ø48 575 to +1 Ø48 575	2
22	4 194 3Ø4	Ø to 4 194 3Ø3	-2 Ø97 151 to +2 Ø97 151	2
23	8 388 6Ø8	Ø to 8 388 6Ø7	-4 194 3Ø3 to +4 194 3Ø3	2
24	16 777 216	Ø to 16 777 215	-8 388 6Ø7 to +8 388 6Ø7	2
25	33 554 432	Ø to 33 554 431	-16 777 215 to +16 777 215	2
26	67 1Ø8 864	Ø to 67 1Ø8 863	-33 554 431 to +33 554 431	2
27	134 217 728	Ø to 134 217 727	-67 1Ø8 863 to +67 1Ø8 863	2
28	268 435 456	Ø to 268 435 455	-134 217 727 to +134 217 727	2
29	536 87Ø 912	Ø to 536 87Ø 911	-268 435 455 to +268 435 455	2
3Ø	1 Ø73 741 824	Ø to 1 Ø73 741 823	-536 87Ø 911 to +536 87Ø 911	2

As an example of the use of this table, suppose you are dealing with integers in the range from -12ØØ to +18ØØ. You would need a segment size of 12 (signed), which gives a range from -2Ø47 to +2Ø47. There would be 5 segments in each variable. Your -define- might look like

```
define segment,dates=v14Ø,12,signed
```

You need not understand the rationale behind this table in order to be able to use segments effectively. An explanation of the underlying "binary" or "base 2" number system and the associated concept of a "bit" are discussed later in an optional section of this chapter.

A common use of segments is as "flags" or markers in a lesson. For example, you might like to keep track of which topics the student has completed or which questions in a drill have been attempted. A segment size of just one is sufficient for such things, with the segment first initialized to zero, then set to one when the topic or question has been covered. The definition might look like this:

```
        define  flags=v2
                segment,flag=flags,1
```

Start by executing "zero flags" to clear all sixty segments in v2. (If you use up to 120 markers you would use "zero flags,2" to clear two variables, each containing 60 segments.) When the student completes the fourth topic you say "calc   flag(4)⇐1" to set the fourth flag. You can retrieve this information at any time to display to the student which topics he has completed.

Although only whole numbers can be kept in segments, it is possible to use the space-saving features of segments even when dealing with fractional numbers. Suppose you have prices of items which in dollars and cents involve fractions, such as $37.65 (37 dollars plus 65 hundredths of a dollar). Assume that $5Ø is the highest price for an item. Simply express the prices in <u>cents</u>, with the top price then being 5ØØØ cents. From the table we see that a segment size of 13 will hold positive integers up to 8191, so we say

```
        define  price=v1    $$ in dollars and cents
                segment,cents=v2,13
                put(i)=cents(i)⇐1ØØprice
                get(i)=price⇐cents(i)/1ØØ
```

A sequence using these definitions might look like

```
        calc    price⇐28.37
        .
        .
        .
        calc    put(16)    $$ equivalent to "cents(16)⇐1ØØprice"
        .
        .
        .
        show    get(16)    $$equivalent to "price⇐cents(16)/1ØØ"
```

The final -show- will put "28.37" on the screen, even though between the "put" and "get" the number was the integer "2837". Notice the unusual "calc   put(16)" which has an assignment (⇐) implicit in the definition of "put". Also notice that the variable "price" is affected as a side-effect of referring to "get". If this is not desired, we could define "get(i)=cents(i)/1ØØ".

As another example of the use of segments with fractional numbers, suppose you have automobile trip mileages up to 1ØØØ miles which you want to store to the nearest tenth-mile (such as 243.8 miles). In this case you must multiply by 1Ø when storing into a segment and divide by 1Ø when retrieving the information. You would use a segment size of 14, since your biggest number is 1ØØØØ. It should be pointed out that rounding to the nearest integer occurs when storing a non-integer value into a segment:

```
        calc    miles⇐539.47
                seg(2)⇐1Ø miles    $$ 5394.7 becomes 5395
                miles⇐seg(2)/1Ø    $$5395/1Ø or 539.5
```

So by going into and out of the segment, the "539.47" has turned into "539.5".

Aside from the restriction to integers, calculations with segmented variables have one further disadvantage: they are much slower than calculations with whole variables. This is due to the extra manipulations the computer must perform to compute which variable contains the Nth segment, and extract or insert the appropriate segment. Segments save space at the expense of time. In many cases this does not matter, but you should avoid doing a lot of segment calculations in a heavily-computational repetitive loop, such as an iterative -do- done ten thousand times!*

Branching within a unit: -branch- and -doto-

All of the branching or sequencing commands discussed so far referred to -unit-s (or -entry-s). It is often convenient to be able to branch within a unit, which is possible with the -branch- command:

```
        unit    somethin
        branch  count-4,5,x,8after
        at      1215
        write   "count" is equal to 4
        5
        do      countit
        8after  count⇐15
```

The tag of the -branch- command is like the tag of a -goto-, except that unit names are replaced by "statement labels". These labels appear at the beginning of statements and must start with a number (Ø through 9) to distinguish them from commands, which start with letters. A statement beginning with a label need not have any tag (as in the line above labeled "5"), but it can have a tag like that of a -calc-, as in the last statement above ("8after count⇐15"). In fact, a labeled statement is essentially a -calc- statement.

* You can now define another kind of segmented variable ("define segment,vertical,...") which is handled much faster.

As with -goto-, "x" in a -branch- means "fall through" to the next statement. The "q" option in a -branch- means "quit doing a -calc-" and proceed to the next non-calc statement. Here is an example:

```
branch  t>15,q,x
calc    a⇐b
        b⇐b+t
8end    t⇐t²
        a⇐a+t
calc    b⇐b³
write   Anything
```

If t is greater than 15, we have a "branch q" all the way to the -write-, which is the first non-calc statement following the -branch-.

The -branch- command is itself a non-calc command:

```
branch  a,q,x
calc    a⇐b
branch  c=d,8i,8j,8k
```

If "a" rounds to a negative integer, the "q" branch is taken to the next -branch- command, bypassing the -calc-.

There is an alternative version of -branch- which is a -calc- as far as a preceding "branch q" is concerned. The word "branch" is placed in the tag:

```
branch  a,q,x
calc    a⇐b
        branch a>c,x,9skip
        c⇐a²
9skip
write   some more
```

The "branch q" takes us to the -write-, since the only things between the first -branch- and the -write- are lines starting with -calc-, blanks ("continued -calc-"), or statement labels (which also are -calc- statements). So -branch- can be either -calc- type or non-calc type. This has certain typographical advantages, permitting a kind of indenting which can make a complicated sequence of -calc- statements more readable. Other than the effect on a preceding "branch q", there is no difference between the two kinds of -branch-.

It is not permissible in a unit to label two statements with the same label (nor can you have two units with the same name in a lesson). On the other hand, since -branch- operates only within a unit and cannot refer to labels in other units, it is all right to use the same label in different units. (Similarly, you can use the same unit name in different lessons.) Note that -entry- is similar to -unit-, so -branch- cannot be used to branch to a label if an -entry- command intervenes.

It is often convenient to use -branch- rather than -goto-. In addition, -branch- requires less computer processing than -goto-, so that heavily computational iterations are better done with -branch- where possible.

Just as -branch- is a fast -goto- within a unit, there is a fast -doto- analogous to -do- for use within a unit:

```
doto    8end,i⇐first,last,incr
calc    a⇐b×sin(5i°)
at      100,200+2a-i
write   T
8end
circle  100,200,300
```

The tag of the -doto- is similar to an iterative -do-, but instead of naming a unit to be done repetitively you name a statement label. For each iteration TUTOR executes statements from the -doto- down to the named statement label. After the last iteration is performed, TUTOR proceeds to the statement which follows the -doto- label (-circle- in the above example).

Just as it is possible to have nested -do- iterations, nested -doto-s can be set up. Here is a comparison of -do- and -doto- for displaying a two-dimensional array:

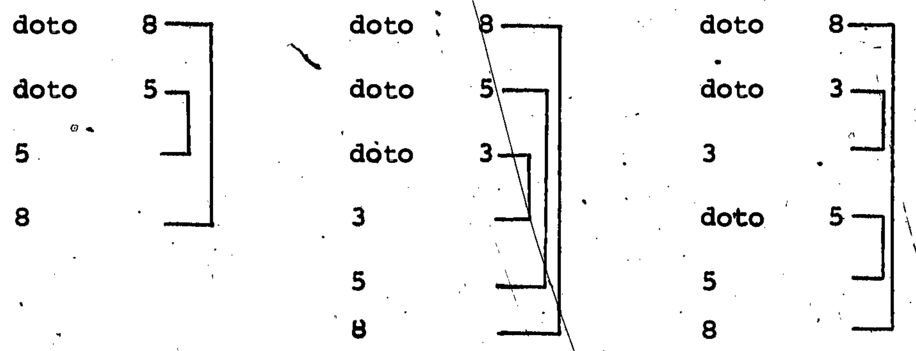
```
          -do-                           -doto-

do      column,i⇐1,3            doto    4,i⇐1,3
unit    column                  doto    4,j⇐1,20
do      rows,j⇐1,20             at      820+10i+100j
unit    rows                    show    scores(i,j)
at      820+10i+100j            4
show    scores(i,j)
```

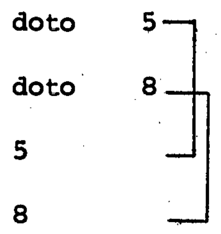
This nested -doto- example has the structure

```
doto    4 ──┐
            │
doto    4 ─┐│
           ││
           ││
4       ───┘┘
```

Other possible structure include these:



Note that in each case the "inner" -doto-s are nested within the "outer" -doto-s. Here is a counter-example which is not permissible:



ILLEGAL!

TUTOR does not permit this kind of structure.

As with -branch-, it is possible to put -doto- in the tag, in which case it is skipped over by a preceding "branch q". Again, the purpose of this kind of indenting is to make some kinds of long continued -calc- statements more readable.

Integer variables and bit manipulation

This section goes much more deeply into the way a computer represents numbers and character strings. Skim this section on the first time through to see whether you will need to study it in detail. You should need this material only if you pack several pieces of data in one variable or if you want to use -calc- operations on character strings.

A variable such as v15Ø can hold a number as big as $1Ø^{322}$(the number 1 followed by 322 zeros!) or a non-zero number as small as $1Ø^{-293}$ (a 1 in the 293rd position after the decimal point!). These huge or tiny numbers may be positive or negative, from $\pm 1Ø^{-293}$ up to $\pm 1Ø^{322}$. Any number held in v15Ø is recorded as sixty tiny "bits" of information. For example, whether the number is positive or negative is one bit of information, and whether the magnitude is $1Ø^{+2ØØ}$ or $1Ø^{-2ØØ}$ is another bit of information. The remaining 58 bits of information are used to specify precisely the number held in v15Ø.

What is a bit? A bit is the smallest possible piece of information and represents a two-way (binary) choice: yes or no (or true or false, or up or down; anything with two possibilities). A number is positive or negative and these two possibilities can be represented by one bit of information. Numbers themselves can represented by bits corresponding to yes or no. Let us see how any number from zero to seven can be represented by three bits corresponding to the yes or no answers to just three questions. Suppose a friend is thinking of a number between zero and seven and you are to determine it by asking the fewest possible questions to be answered yes or no. Suppose the friend's number is 6:

a)	Is it as big as 4?	Yes.
b)	Is it as big as 4+2?	Yes.
c)	Is it as big as 4+2+1?	No.

From this you correctly conclude that his number is 6. You determined that his number was made up of a 4, a 2, and no 1. You might also say that his number can be represented by the sequence "yes,yes,no"!

As another example, try to guess a number between zero and 63 chosen by the friend. Suppose it is 37:

a)	Is it as big as 32?	Yes.
b)	Is it as big as 32+16?	No.
c)	Is it as big as 32+8?	No.
c)	Is it as big as 32+4?	Yes.
d)	Is it as big as 32+4+2?	No.
e)	Is it as big as 32+4+1?	yes.

So the number is 37, or perhaps "yes,no,no,yes,no,yes"! Try this questioning strategy on any number from zero to 63 and you will find that six questions are always sufficient to determine the number. The strategy depends on cutting the unknown range in two each time (a so-called "binary chop").

Conversely, any number between zero and 63 can be represented by a sequence of yes and no answers to six such questions. What number is represented by the sequence

yes,yes,no,yes,no,yes?

This number must be built up of a 32, a 16, no 8, a 4, no 2, and a 1. 32+16+4+1 is 53, so the sequence represents the number 53.

Because a yes or no answer is the smallest bit of information we can extract from our friend, we say any number between zero (six nos) and 63 (six yeses) can be represented by six bits. If on the other hand we know the number is between zero and seven, three bits are sufficient to describe the number fully. Similarly, numbers up to 15 (2^4-1) can be expressed with four bits, and numbers up to 31 (2^5-1) with five bits. Each new power of two requires another bit because it requires another yes/no question to be asked.

This method of representing numbers as a sequence of bits, each bit corresponding to a yes or no, is called "binary notation" and is the method normally used by computers. Whether a computer bit represents yes or no is typically specified by a tiny electronic switch being on or off, or by a tiny piece of iron being magnetized up or down. A TUTOR variable contains <u>sixty</u> bits of yes/no information and could therefore be used to hold a <u>positive integer</u> as big as $(2^{6\emptyset}-1)$, which is approximately $1\emptyset^{18}$, or 1 followed by 18 zeros! What do we do about <u>negative</u> integers? Instead of using all sixty bits we could give up one bit to represent whether the number is positive or negative (again, a two-way or binary bit of information) and just use 59 bits for the magnitude of the number. In this way we could represent positive or negative <u>integers</u> up to $\pm(2^{59}-1)$, which is approximately plus or minus one-half of $1\emptyset^{18}$.

But what do we do about bigger numbers, or numbers such as 3.782 which are not integers? The scheme used is analogous to the scientific notation used to express large numbers: $6.\emptyset2\times1\emptyset^{23}$ is a much more compact form than 6∅2 followed by 21 zeros, and it consists of two essential pieces--the number 6.∅2 and the <u>exponent</u> or <u>power</u> of ten (23). Instead of using 59 bits for the number, we use only 48 bits and use 11 bits for the exponent. Of these 11 bits, one is used to say whether the exponent is positive or negative (the difference between $1\emptyset^{+6}$, a million, and $1\emptyset^{-6}$, one-millionth). The remaining ten bits are used to represent exponents as big as one thousand ($2^{1\emptyset}-1$ is 1∅23, to be precise). The exponent is actually a power of two rather than ten, as though our scientific notation for the number 4∅ were written as 5×2^3 instead of $4\times1\emptyset^1$. That is, instead of expressing the number forty as $4\times1\emptyset^1$, we express it as 5×2^3, putting the 5 in our 48-bit number and the 3 in the 11-bit exponent storage place. In this way we split up the 6∅ bits as

1 bit for positive or negative number
1 bit for positive or negative exponent
1∅ bits for the power of two
48 bits for the number

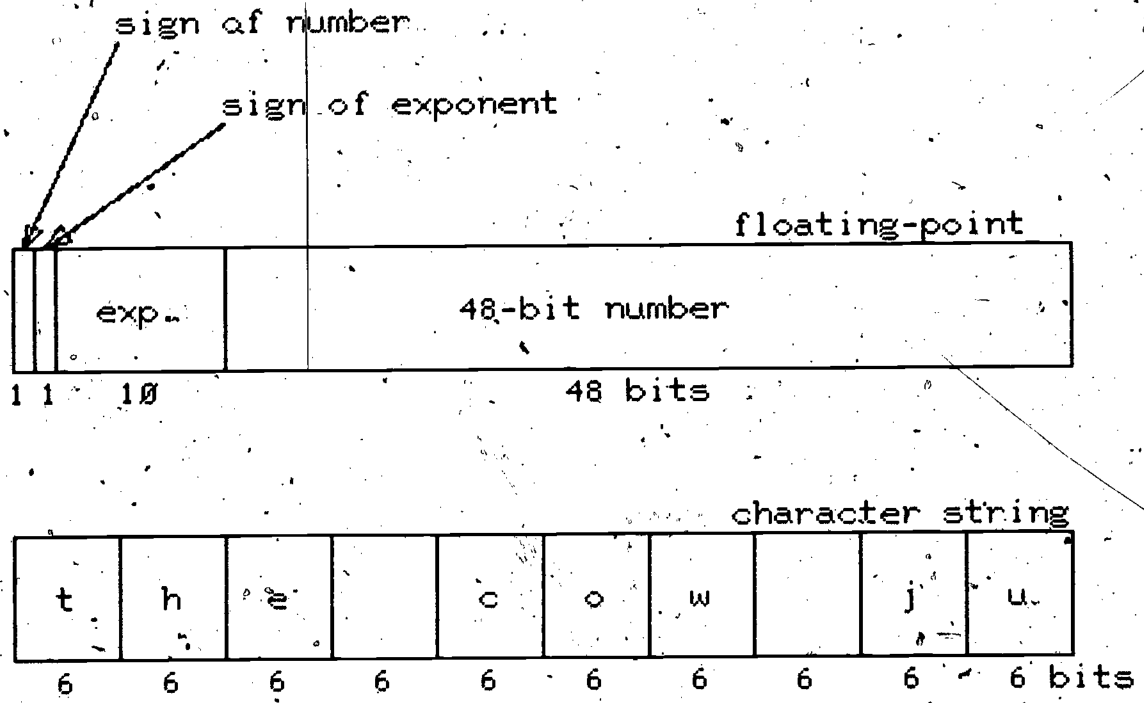
The 48-bit number will hold an integer as big as $(2^{48}-1)$, which is about $2.5\times1\emptyset^{14}$. If we wish to represent the number 1/4, the variable will have a number of 2^{47} and an exponent of -49:

$$2^{47}\times2^{-49}= 2^{-2}= 1/4$$

That is, the 48-bit number will hold a large integer, 2^{47}, and the exponent, or power of 2, will be -49. The complicated format just described is that used by the PLATO computer when we calculate with variables v1 through v15∅. It automatically takes care of an enormous range of numbers by separating each number into a 48-bit number and a power of two. This format is called "fractional" or "floating-point" format because non-integral values can be expressed and the position of the decimal point floats automatically right or left as operations are performed on the variable.

Sometimes this format is not suitable, particularly when dealing with strings of characters. The -storea- command and -pack- commands place ten alphanumeric characters into each variable or "word" (a computer variable is often called a "word" because it can contain several characters). We simply split up the sixty bits of the word into ten characters of six bits each, six bits being sufficient to specify one of 64 possible characters, from character number zero to character number 63 (2^6-1). In this scheme character number 1 corresponds to an "a", number 2 to a "b", number 26 to a "z", number 27 to a "Ø", number 28 to a "1", etc. A capital D requires two 6-bit character slots: one for a "shift" character (which happens to be number 56) and one for a lower-case "d" (number 4). The -showa- command takes such strings of 6-bit character codes and displays the corresponding letters, numbers, or punctuation marks on the student's screen.

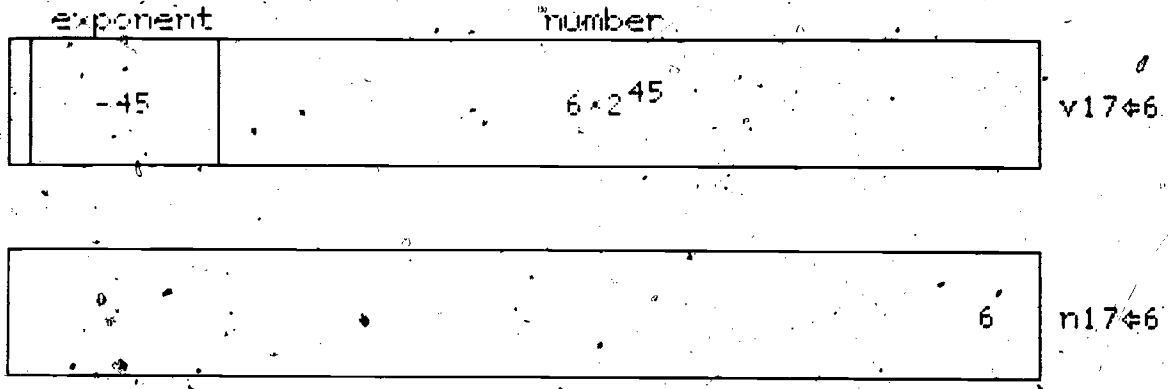
Ridiculous things happen if a -showa- command is used to display a word which contains a floating-point number. The two sign bits (for the number and for the exponent) and the first four bits of the the exponent make up the first 6-bit character code! The last six bits of the exponent are taken as specifying the second 6-bit code. Then the remaining 48 bits are taken as specifying eight 6-bit character codes. Small wonder that using a -showa- on anything other than character strings usually puts gibberish on the screen. On the other hand, using a -show- with a character string gives nonsense: the floating-point exponent gets made up out of pieces of the first and second 6-bit character codes, the 48-bit number comes from the last eight character codes, and whether the number and the exponent are positive or negative is determined by the first two bits of the first character code!



So far we have kept numerical manipulations (-calc-, -store-, -show-) completely separate from character string manipulations (-storea-, -showa-). The reasons should now be clear. It is nevertheless sometimes advantageous to be able to use the power of -calc- in manipulating character strings and similar sequences of bits. For such manipulations we would like to notify TUTOR not to pack numbers into a variable in the useful but complicated floating-point format. This is done by referring to "integer variables"

n1,n2,n3 ---------------n149,n150

The integer variable n17 is the same storage place as v17, but its internal format will be different. If we say "calc v17⇐6", TUTOR will put into variable number 17 the number 6, expressed as 6×2^{45} with an exponent of -45, so that the complete number is $6\times2^{45}\times2^{-45}$, or 6. If on the other hand we say "calc n17⇐6", TUTOR will just put the number 6 into variable number 17. Since the number 6 requires only three bits to specify it, variable 17 will have its first 57 bits unused, unlike the situation when we refer to the 17th variable as v17, in which case both the exponent and the magnitude portions of the variable contain information.



Consider the following sequence:

```
calc    n17⇐6
.
.
.
at      1223
showa   n17,10
```

This will cause an "f" (the 6th letter in the alphabet) to appear on the screen at location 1223. The first 9 character codes in n17 are zero, and these zero or "null" codes have no effect on the screen or screen positioning. Indeed, a "showa n17,9" would display nothing since the "6" is in the tenth character slot. If we use "show n17", we will just see a "6" on the screen. The integer format of n17 alerts -show- not to expect a floating-point format.

If you say "calc n23⇐5.7", variable n23 will be assigned the value 6: rounding is performed in assigning values to integer variables. If truncation is desired, use the "int" function: "n23⇐int(5.7)" will assign the integer part (5) to n23. Indexed integer variables are written as "n(index)" in analogy with "v(index)".

The -showa- and -storea- commands may be used with either v-variables or n-variables. These commands simply interpret any v- or n-variable as a character string. This is the reason why we were able to use -showa- and -storea- without discussing integer variables.

It is possible to shift the bits around inside an integer variable. In particular, a "circular left shift", abbreviated as cls, will move bits to the left, with a wrap-around to the right end of the variable:

```
calc     n17⇐6 $cls$ 54
.
.
.
at       1223
showa    n17,1   $$show one character
```

will display an "f" even though the -showa- will display only the first character, because the "6" has been shifted left 54 bit positions (9 six-bit character positions). A circular left shift of 54 may also be thought of as a right circular shift of 6 because of the wrap-around nature of the circular shift.

We have been using "n17" as an example, but of course we should be writing "inum" or some such name, where we have used a -define- to specify that "inum=n17". For the remainder of this chapter we revert, therefore, to the custom of referring to variables (v or n) by name rather than number. Also, if we want the character code corresponding to the letter "f" we should use "f" rather than 6:

```
calc     inum⇐"f" $cls$ 54
```

is equivalent to but much more readable than

```
calc     n17⇐6 $cls$ 54.
```

The quotation marks can be used to specify strings of characters. For example,

```
calc     inum⇐"cat"
```

will put these numbers in inum:

null	null	null	null	null	null	null	c	a	t
0	0	0	0	0	0	0	3	1	20

A "showa inum,1Ø" will display "cat". Notice particularly that using quotes in a -calc- to define a character string puts the string at the right ("right adjusted"), whereas the -storea- and -pack- commands produce left-adjusted character strings. It is possible to create left-adjusted character strings by using single quote marks: inum⇐'cat' will place the "cat" in the first three character positions rather than the last three.

Let us now return to our early example of the number 37 expressed as the sequence of six bits "yes,no,no,yes,no,yes". If we let 1 stand for "yes", and Ø for "no", we might write this sequence as

1ØØ1Ø1

which stands for

$$(1\times32)+(\emptyset\times16)+(\emptyset\times8)+(1\times4)+(\emptyset\times2)+(1\times1) = 32+\emptyset+\emptyset+4+\emptyset+1 = 37$$

or even more suggestively

$$(1\times2^5)+(\emptyset\times2^4)+(\emptyset\times2^3)+(1\times2^2)+(\emptyset\times2^1)+(1\times2^\emptyset) = 32+\emptyset+\emptyset+4+\emptyset+1 = 37$$

(Note that $2^\emptyset$ equals 1.) Writing the sequence in this way is analogous to writing 524 as

$$(5\times1\emptyset^2)+(2\times1\emptyset^1)+(4\times1\emptyset^\emptyset) = 5\emptyset\emptyset+2\emptyset+4 = 524$$

In other words, when we write 524 we imply a "place notation" in base 1Ø such that each digit is associated with a power of 1Ø: $5\times1\emptyset^2$, $2\times1\emptyset^1$, $4\times1\emptyset^\emptyset$. Similarly, rewriting our yes and no sequences as 1 and Ø sequences we find that the string of ones and zeros turns out to be the place notation in base 2 for the number being represented.

Here are some examples. ($1\emptyset\emptyset1_2$ means 1ØØ1 in base 2.)

$$1\emptyset\emptyset1_2 = 2^3+2^\emptyset = 8+1 = 9$$

$$11\emptyset\emptyset_2 = 2^3+2^2 = 8+4 = 12$$

$$11\emptyset1\emptyset1_2 = 2^5+2^4+2^2+2^\emptyset = 32+16+4+1 = 53$$

$$1\emptyset\emptyset\emptyset\emptyset\emptyset1_2 = 2^6+2^\emptyset = 64+1 = 65$$

This base 2 (or "binary") notation can be used to represent any pattern of bits in an integer variable, and with some practice you can mentally convert back and forth between base 1Ø and base 2. This becomes important if you perform certain kinds of bit manipulations.

An important property of binary representations is that shifting left or right is equivalent to multiplying or dividing. Consider these examples:

```
                                      ←shift left 2 places
        9 $cls$ 2 = 1ØØ1₂ $cls$ 2 = 1ØØ1ØØ₂ = 36
        (left shift 2 is like multiplying by 2² or 4)

                          ←shift left 3 places
        9 $cls$ 3 = 1ØØ1ØØØ₂ = 72
        (left shift 3 like multiplying by 2³ or 8)
```

So a left shift of N bit positions is equivalent to multiplying by 2^N. A <u>right</u> shift of N bit positions is equivalent to <u>division</u> by 2^N (assuming no bits wrap around to the left end in a cls of 6Ø-N). There exists an "arithmetic right shift", ars, which is not circular but simply throws away any bits that fall off the right end of the word:

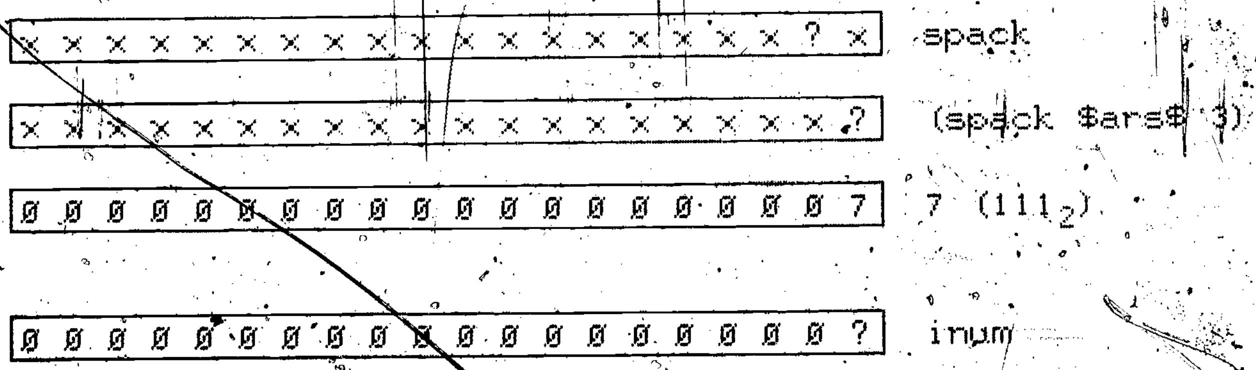
```
                                             thrown away
        9   $ars$ 3 = 1ØØ1₂  $ars$ 3 = 1ØØ1 = 1.
```

This corresponds to a division by 2^3, with truncation ($9/2^3 = 9/8$ which truncates to 1).

A major use of the 6Ø bits held in an integer variable is to pack into one word many pieces of information. For example, you might have 6Ø "flags" set up or down (1 or Ø) to indicate 6Ø yes or no conditions, perhaps corresponding to whether each of 6Ø drill items has been answered correctly or not. Or you might keep fifteen 4-bit counters in one word: each 4-bit counter could count from zero as high as 15 (2^4-1) to keep status on how well the student did on each of fifteen problems. Ten bits is sufficient to specify integers as large as 1Ø23: you could store six 1Ø-bit baseball batting averages in one word, with suitable normalizations. Suppose a batting average is .324. Multiply by a thousand to make it an integer (324) and store this integer in one of the 1Ø-bit slots. When you withdraw this integer, divide is by a thousand to rescale it to a fraction (.324). When we discussed arrays we had exam scores ranging from zero to 1ØØ. The next larger power of two is 128 (2^7), so we need only 7 bits for each integer exam score. Eight such 7-bit quantities could be stored in one 6Ø-bit word.

How do you extract a piece of information packed in a word? As an example, suppose you want three bits located in the 19th of twenty 3-bit slots of variable "spack":

```
        inum⇐(spack $ars$ 3) $mask$ 7
```



x x x x x x x x x x x x x x x x x x ? x
spack
x x x x x x x x x x x x x x x x x x x ?
(spack ars 3)
Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø 7
7 (111₂)
Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø ?
inum

The number 7 is 111_2 (base 2: 4+2+1), so it is a 3-bit quantity with all three bits "set" or "on" (non-zero). The \$mask\$ operation pulls out the corresponding part of the other word, the 3-bit piece we are interested in. In an expression (x \$mask\$ y) the result will have bits set (1) only in those bit positions where both x and y have bits set. In those bit positions where either x or y have bits which are "reset" or "off" (Ø), the \$mask\$ operation produces a Ø. We could also have used a "segment" definition to split up the word into 3-bit segments.

A 4-bit mask would be 15 (1111_2); a 5-bit mask 31 (11111_2). (Again, "segment" definitions of 4 or 5 bits could be used). You might even need a mask such as $11\varnothing111_2$, which is 55; it will extract bits located in the five bit positions where $11\varnothing111_2$ has bits set. There should be a simpler way of writing down numbers corresponding to particular bit patterns. Certainly reading the number 55 does not immediately conjure up the bit pattern $11\varnothing111_2$!

A compact way of expressing patterns of bits depends on the fact that each set of three bits can represent a number from Ø to 7:

$$\begin{tikzcd}
 & 55 = 11\varnothing111_2 \arrow[dl] \arrow[dr] & \\
11\varnothing_2 = 4+2+\varnothing = 6 \arrow[dr] & & 111_2 = 4+2+1 = 7 \arrow[dl] \\
 & \underset{\text{(base 8)}}{67_8} = 6\times 8^1 + 7\times 8^\varnothing = 48+7 = \underset{\text{(base 1Ø)}}{55_{1\varnothing}} &
\end{tikzcd}$$

Just as each digit in a decimal number (base 1Ø) runs from Ø to 9, so in an octal number (base 8) the individual numerals run from Ø to 7. Octal numbers are useful only because they represent a compact way of expressing bit patterns. With practice one converts between octal and base 2 instantaneously, and between base 8 and base 1Ø somewhat slower!

base 1Ø	base 8		base 2		
Ø	Ø		Ø	or	ØØØ
1	1		1	or	ØØ1
2	2	These	1Ø	or	Ø1Ø
3	3	should	11	or	Ø11
4	4	be	1ØØ		
5	5	memorized	1Ø1		
6	6		11Ø		
7	7		111		
8	1Ø		1ØØØ		
9	11		1ØØ1		
1Ø	12		1Ø1Ø		
11	13		1Ø11		
12	14		11ØØ		
13	15		11Ø1		

The conversion between base 8 and base 2 is a matter of memorizing the first eight patterns, after which translating 1101011011101_2 to octal is simply a matter of drawing some dividers every three bits:

```
1|101|011|011|101
1  5   3   3   5     =    15335₈
```

What is 15335_8 in base 1Ø?

8^4	8^3	8^2	8^1	$8^Ø$
4Ø96	512	64	8	1
1	5	3	3	5

$= 1\times4Ø96+5\times512+3\times64+3\times8+5 = 5853_{1Ø}$

How about the octal version of the number 79? The biggest power of 8 in 79 is 8^2 (64), and 79 is 15 more than 64. In turn, 15 is $1\times8^1+7\times8^Ø$, so

$$79_{1Ø} = 1\times64+1\times8+7\times1 = 1\times8^2+1\times8^1+7\times8^Ø = 117_8$$

Luckily, in bit manipulations the conversions between base 2 and base 8 are more important than the harder conversions between base 8 and base 1Ø.

To express an octal number in TUTOR, use an initial letter "o":

```
x $mask$ o37
```

will extract the right-most 5 bits from x, because $o37 = 37_8 = Ø11111_2$, which has 5 bits set. Naturally a number starting with the letter "o" must not contain 8's or 9's!

You can display an octal number with a -showo- command (show octal):

```
showo   39
```

will display "ØØØØØØØØØØØØØØØØØØ47" on the screen ($39_{1Ø}=47_8$). The default format is twenty (3-bit) octads, corresponding to a whole 6Ø-bit word.

```
showo   39,4
```

will display "ØØ47", showing just four octads.

Now that we have discussed the octal notation, it is possible to point out what happens to negative numbers:

```
showo   -39
```

will display "777777777777777777730"! A negative number is the "complement" of the positive number--binary 1's are changed to Ø's and binary Ø's are changed to 1's. In octal, the complement of Ø is 7 ($ØØØ_2 \rightarrow 111_2 = 7_8$), and the complement of 7_8 is $Ø_8$. In the example shown, octal 47_8 is $1ØØ111_2$.

whose complement is $\emptyset 11 \emptyset\emptyset\emptyset_2$, or $3\emptyset_8$. Notice in particular that the left-most bit (the "sign" bit) of a negative number is always set. In order that a negative number stay negative upon performing an "arithmetic right shift", all the left-most bits are set. So

```
                o4ØØØØØØØØØØØØØØØ3242 $ars$ 6
     yields     o774ØØØØØØØØØØØØØØØ32.
```

Only the sign bit was set among the left-most bits before the shift (o4Ø is $1\emptyset\emptyset\emptyset\emptyset\emptyset_2$), but after the shift the first seven bits are all set. The "circular left shift", cls, does not do anything special with the sign bit.

It is interesting to see the bits set for floating-point numbers:

```
        :
        :
        calc    v1⇐3
        at      1215
        write   pos=⊲o,v1⊳    $$ o for -showo-
                neg=⊲o,-v1⊳
```

will make this display:

```
                pos = 17216ØØØØØØØØØØØØØØØ
                neg = 6Ø5717777777777777777
```

Note that the negative number is the complement of the positive. The 48-bit magnitude (6ØØØØØØØØØØØØØØØ) represents a huge integer (6×2^{45}). The eleven bits between the sign bit and the 48-bit magnitude give the power of two (-46) by which the magnitude is to be scaled ($3 = 6\times2^{45}\times2^{-46} = 6\times2^{-1} = 3$). A bias of $2\emptyset\emptyset\emptyset_8$ is added to the correct exponent (-46, or -56_8) to give an eleven-bit exponent of 1721_8. Exponents less than $2\emptyset\emptyset\emptyset_8$ represent negative powers; exponents greater than $2\emptyset\emptyset\emptyset_8$ represent positive powers.

We have encountered octal numbers (e.g., o327) which can be shifted left (cls) and right (ars) and complemented (by making negative). Pieces can be extracted with a $mask$ operation. Additional bit operations are $union$, $diff$, and "bitcnt". The "bitcnt" function gives the number of bits set in a word: bitcnt(o25) is 3, because o25 is $\emptyset1\emptyset1\emptyset1_2$, which has 3 bits set; bitcnt(-o25) is 57, since the complement will have only 3 of 6Ø bits not set; bitcnt(Ø) is Ø. Like $mask$, $union$ and $diff$ operate on the individual bit positions, with all 6Ø done at once:

x $mask$ y	produces a 1 only where <u>both</u> x and y have 1's.
x $union$ y	produces a 1 where <u>either</u> x <u>or</u> y <u>or</u> <u>both</u> have 1's.
x $diff$ y	produces a 1 only where x and y <u>differ</u>.

Note that \$union\$ might be called "merge", since 1's will appear in every bit position where either x or y have bits set. The \$diff\$ operation might also be referred to as an "exclusive" union, since it will merge bits except for those places where both x and y have bits set.

While \$mask\$ can be used to extract a piece of information from a word, a \$mask\$ that includes all but that piece followed by a \$union\$ can be used to insert a new piece of information.

Byte manipulation

The most common use of bit manipulations is for packing and unpacking "bytes" consisting of several bits from words each of which contain several bytes. This can lead to major savings in space. If an exam score lies always between Ø and 1ØØ, only seven bits are required to hold each score, since (2^7-1) is 127. Another way to see this is to write the largest 7-bit quantity: $1111111_2 = 177_8 = 1\times8^2+7\times8^1+7\times8^Ø = 64+56+7 = 127$. This is one less than $2ØØ_8$, which requires an eighth bit. We can fit eight 7-bit bytes into each 6Ø-bit word. Happily, TUTOR will do the bookkeeping for you, as we saw earlier:

```
          define   segment,scores=n31,7
```

This definition makes it possible to work with this "segmented" array as though it were an ordinary array:

```
          calc     ss⇐scores(3)
                   scores(17)⇐83
                       etc.
```

These refer to the 3rd and 17th bytes. The first eight 7-bit bytes reside in n31, with the last 4 bits unused. The next eight bytes are in n32, etc. The 17th byte is the first 7-bit byte in n33.

Just as we effectively give up one bit of a 6Ø-bit word in order to have negative as well as positive numbers, so it is possible to have both positive and negative numbers stored in a segment array:

```
          define   segment,temp=v52,8,signed
          .
          .
          .
          calc     temp(23)⇐-95
```

With 8-bit bytes we can have numbers in the range of ±127. The word "signed" may be abbreviated by "s".

Now that you understand the bit structure of a variable, you should be able to understand the table given earlier of segment ranges and number of segments per variable. Look at the table now and see whether you can check the entries in the table.

Alphanumeric to numeric: the -compute- command

The -store- command analyzes the judging copy of the student's response character string and produces a numerical result. This is actually a two-step process. First the character string is "compiled" into basic computer instructions and then these machine instructions are "executed" to produce the numerical result. During the compilation process the "define student" definitions and the built-in function definitions (sin, cos, arctan, etc.) are used to recognize the meaning of names appearing in the character string. Numbers expressed as alphanumeric digits are converted to true numerical quantities. For example, the character string 49 becomes a number by a surprisingly indirect process. The character code for "4" is 31 since "z" is 26, "Ø" is 27, etc. The character code for "9" is 36. The number expressed by typing 49 is obtained from the formula

```
1Ø(31-27)+(36-27)  or  1Ø("4"-"Ø")+("9"-"Ø")
        1Ø(4)+(9)
          4Ø+9
           49
```

For these and similar reasons, the compilation process is ten to a hundred times slower than the execution process! Therefore, TUTOR attempts to compile the student's response only once, while the resulting machine instructions may be used many times.

The first -store-, -ansv-, -wrongv-, -storeu-, -ansu-, or -wrongu- command encountered during judging triggers compilation. All these commands following the first one simply reuse the compiled machine instructions. If a -bump- or -put- makes any changes in the judging copy, a following -store- or related command will have to recompile. Similarly, a "judge rejudge" will force recompilation by any of these commands. Note that re-execution is always performed even if recompilation isn't, because the student might refer to defined variables whose values have been altered.

While -store- will compile and execute from the judging copy, the regular -compute- command will compile and execute from any stored character string:

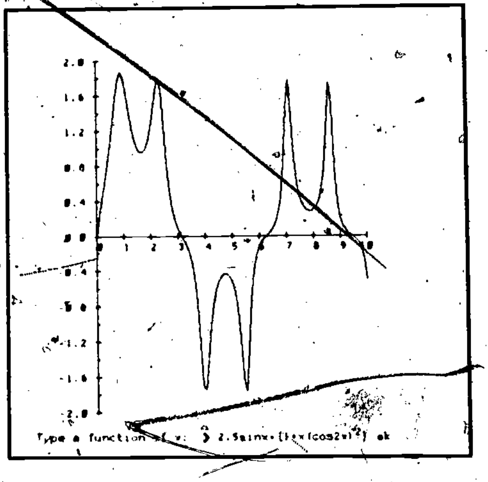
```
compute result,string,#characters,pointer
```

For example,

$$\texttt{compute }\underset{\swarrow\ \text{return numerical result}}{\texttt{v35}}\texttt{,}\underbrace{\texttt{v2,v1}}_{\text{character string}}\texttt{,}\underset{\searrow\ \text{pointer to machine instructions}}{\texttt{v22}}$$

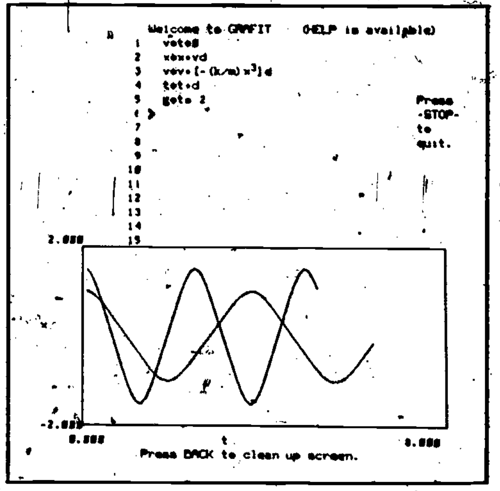
After compilation, the "pointer to machine instructions" contains the location of the machine instructions in a special -compute- storage area. You must zero the pointer at first to force compilation. TUTOR will then set the pointer appropriately so that re-executions of the -compute- command can simply re-execute the saved machine instructions. Here is a unit which permits the student to plot functions of interest to him:

```
define   student
         x=v1
define   ours,student
         result=v2,string=v3,point=v35
origin   1ØØ,25Ø
bounds   Ø,-2ØØ,3ØØ,2ØØ
scalex   1Ø
scaley   2
*
unit     graph
next     graph
back     graph
axes              $$ display the axes
labelx   1
labely   Ø.2
at       31Ø5
write    Type a function of x:
arrow    where+2
storea   string,jcount
ok
calc     x⇐point⇐Ø
compute  result,string,jcount,point
goto     formok,x,badform
locate   Ø,result          $$ draw from here
doto     8plot,x⇐.1,1Ø,.1
compute  result,string,jcount,point
goto     formok,x,badform
gdraw    ;x,result
8plot
*
unit     badform
at       32Ø7
writec   formok,...        $$ tell what's wrong
judge    wrong
```



Different functions can be superimposed by changing the response instead of pressing NEXT or BACK. The first -compute- in this unit calculates the value of the student's function for x equal to zero. The -locate- command positions us at location (Ø, result) so that the first -gdraw- will draw a line starting at that point. The system variable "formok" has the value -1 if compilation and execution succeed; Ø if compilation succeeds but execution fails due to such errors as trying to take the square root of a negative number; and various positive integral values for various compilation errors (missing parentheses, unrecognized variable names, etc.).

As another example, the PLATO lesson "grafit" (written by Bruce Sherwood) permits the student to write up to fifteen statements in the grafit language and execute his program to produce graphical output:



```
             Welcome to GRAFIT    (HELP is available)
    1   v⇐0
    2   x⇐x+vd
    3   v⇐v+[-(k/m)x³]d
    4   t⇐t+d
    5   goto 2
    6 >                                              Press
    7                                                -STOP-
    8                                                to
    9                                                quit.
   10
   11
   12
   13
   14
2.000  15

-2.000
  0.000               t                          8.000
        Press BACK to clean up screen.
```

This student's program calculates the motion of a mass oscillating on the end of a non-standard spring. The two curves are the superposition of running the program twice with different values of the parameters. The heart of this lesson is a loop through a -compute- command with string, character count, and point all being indexed variables. The index is the line number, from 1 to 15. Each student response is analyzed using a -match- command looking for keywords such as "goto". Then the rest of the response is filed away with a -storea- into the string storage area corresponding to that line number. The 15 pointer variables are zeroed in the ieu (initial entry unit) to insure that when the student returns to a PLATO terminal after several days TUTOR won't be confused over whether the strings have been recently compiled or not. Also, whenever the student changes one of his statements, the corresponding pointer is zeroed in order to force recompilation of the altered character string. The student can press DATA to initialize parameters, LAB to specify what variable to plot against what variable, and HELP for a description of the grafit language. The "student" define set defines all 26 letters as variables the student can use.

Built-in definitions take precedence in a -compute-, so that sin(θ) is taken to mean the sine of θ, not s×i×n×(θ), even if s, i, and n have all been defined in the student set. This is true also for -store- and related commands. However, author's own definitions take precedence in a

-calc-, so that within a lesson an author can override the normal meaning of "sin" or "wherex". Other differences already mentioned were that authors must use explicit multiplication between names of variables or functions, and authors must use parentheses around function arguments. While authors are discouraged from using primitive names such as v47 except in a -define- statement, students are not permitted to use primitives at all. This is done to protect the author's internal information. Similarly, students cannot use the assignment symbol (⇐), except in a -compute-, unless there is a "specs okassign".

It should be mentioned that while -compute- converts alphanumeric information into a numerical result, there is an -itoa- command that can be used to convert an integer to an alphanumeric character string.

The -find- command

The -search- command is character-string oriented and will locate 'dog' even across variable or word boundaries: the d might be at the end of one word and the og at the beginning of the next word. The -find- command, in contrast, is word oriented. It will find which word contains a certain number or character string:

```
find     372,n1,5Ø,n125 ----------> return the
         ↙      ↓    ↘              location
                starting  looking
     look       at n1     through
     for 372              5Ø words
```

If n1 contains 372, n125 will return the value Ø; if n2 is the first word which contains 372, n125 will be 1, etc. If none of the 5Ø words contains 372, n125 will be set to -1. Notice that in -search- the return is 1, not Ø, if the string is found immediately. This is due to the fact that in character strings we start numbering with character number 1. On the other hand, here the first word is n(1+Ø).

Do not use v-variables in the first two arguments of -find- because -find- makes its comparisons by integer operations. The first argument can be a character string such as 'dog' or "dog". You can look at every 3rd word by specifying an optional increment:

```
find     "cat",n1,5Ø,n125,3
                          └┘
                       optional
```

This will look for "cat" in n1, n4, n7, etc., and n125 would be returned Ø, or 3, or 6, etc. Negative increments can be used to search backwards from the end of the list.

You can also specify that a "masked equality search" be made:

```
        find     "cat",n1,5ø,n125,1,o777700
                                 \_/       mask
                            not optional
```

In this case n125 will be zero if [(n1 $diff$ "cat") $mask$ o7777øø] is zero. The mask specifies that only a part of the word will be examined. The increment must be specified, even if it is one, to avoid ambiguity.

There is a -findall- command which will produce a list of all the locations where something was found, rather than producing locations one at a time.

The -exit- command

Suppose you are seven levels deep in -do-s. That is, you have encountered seven nested -do- statements on the way to the present unit. The statement "exit 2" will take you out two levels. The next statement to be executed is the statement which follows the sixth -do-. A blank -exit- command (blank tag) takes you immediately to the statement following the first -do-. Such operations are occasionally useful. Notice that encountering a unit command at the end of a done subroutine will cause an automatic "exit 1".

X. Common variables

The -common- command

The "student variables" v1 through v15Ø are associated with the individual student. It is possible to use "common variables" which are common to all those students studying a particular lesson. These common variables can be used to send messages from one student to another, to hold a bank of data used by all the students, to accumulate statistics on student use of the lesson, to contain test items in a compact, standardized form, etc.

As a first example of the use of common, let's count the number of students who have entered our lesson. We will also count how many of these students are female:

```
common    2                                   $$ two common variables
define    total=vc1,females=vc2
*
unit      ask
calc      total⇐total+1
at        1215
write     Are you a female?
arrow     1415
answer    yes
calc      females⇐females+1
answer    no
no
write     Yes or no, please!
endarrow
at        1615
write     There are <s,total> students, of whom
          <s,females> are female.
```

The -common- command tells TUTOR to set up two common variables, vc1 and vc2, which we have defined as "total" and "females". These common variables are automatically initialized to zero before the first student enters this lesson. The first student increments "total" to one ("calc   total⇐total+1") and may also increment "females". The second student to enter the lesson causes "total" to increase to two and may also change "females". Each student is shown the present values of "total" and "females", which depend on what other students are doing. We must use common variables vc1 and vc2 rather than the student variables v1 and v2 because the student variables cannot be directly affected by actions of other students. Another way to see this is to point out that when there are five students in this lesson, they share a single vc1 and a single vc2, whereas they each have their own v1 and their own v2: there are five v1's and five v2's but only one vc1 and vc2.

Integer common variables are nc1, nc2, etc., and indexed common variables are written as vc(index) or nc(index).

The statement "common 2" tells TUTOR to associate a two-word set of common variables with this lesson. For reference purposes it is good style to place the -common- command near the beginning of the lesson. There can be only one -common- statement in a lesson. Like -define-, -vocab-, and -list-, the -common- command is not executed for each student: rather, when TUTOR is preparing the lesson for the first student who has requested it, a set of common variables is associated with the lesson and all these common variables are initialized to zero. Additional students entering the lesson merely share the common variables previously set up.

Suppose a class of fourteen students uses our lesson from 1Ø A.M. to 11 A.M. The fourteenth student comes at 1Ø:Ø5 and gets a message on the screen saying "There are 14 students, of whom 8 are female". As long as the lesson is in active use, each new student who enters the lesson increases "total" (vc1). However, when all the students leave at 11:ØØ, the lesson is no longer in active use and will eventually be removed from active status to make room for other lessons. When another class comes at 3:ØØ P.M., the lesson is not in active use and TUTOR must respond to the first student's request for the lesson by preparing the lesson for active use. In the preparation process the statement "common 2" tells TUTOR to set up two common variables and initialize them to zero. The first student to enter the lesson at 3:ØØ is told "There are 1 students, of whom 1 are female". She is not told "There are 15 students, of whom 9 are female", despite the fact that the previous student (at 1Ø:Ø5 that morning) had been told there were 14 students, 8 female. The "common 2" statement will cause the common variables to be zeroed every time the lesson is prepared for active use.

The type of common which is set up by the statement "common 2" is called a <u>temporary</u> common. It lasts only as long as the lesson is in active use, and its contents are initialized to zero whenever the lesson is moved from inactive to active status. Temporary common can be used for such things as telling the students how many students are present, what their names are, and whether a student who has finished a particular section of the lesson is willing to leave his terminal to help a student who is having difficulties. Messages can be sent from one student to another through a temporary common: just store the message in the common area with an identifying number so that the appropriate student can pick up the message and see it with a -showa-. The lesson simply checks occasionally for the presence of a message.

When a student signs out you usually want to change the temporary common in some way. For example, if you are keeping a count of the number of students presently using the lesson, you increase the count by one when a student signs in and you decrease the count by one when the student leaves. The -finish- command lets you define a unit to be executed when the student presses shift-STOP to sign out:

```
        finish  decrease
        .
        .
        unit    decrease
        calc    count⇐count-1
```

In this case unit "decrease" will be done each time a student signs out. Normally the -finish- command should be put in the ieu (initial entry unit). As with -imain-, the pointer set by the -finish- command is not cleared at each new main unit. A later -finish- command overrides an earlier one, and "finish q" or a blank -finish- statement will clear the pointer. As with all unit pointer commands, -finish- can be conditional. Only a limited amount of processing is permitted in a -finish- unit to insure that the student can in fact sign out promptly.

We can keep a permanent, on-going count of students who enter the lesson by using a permanent common. Instead of writing "common 2", we write "common italian,counts,2", where "italian" is the name of a permanent lesson storage space and "counts" is the name of a common block stored there. This is the same format used for character sets (the -charset- command) and micro tables (the -micro- command). When the common block is first set up in the lesson space, its variables are initialized to zero. Let's suppose that the fourteen students who come in at 1ØːØØ A.M. are the very first students ever to use our lesson. The statement "common italian,counts,2" will cause TUTOR to fetch this (zeroed) common block from permanent storage. As before, the fourteenth student arrives at 1Ø:Ø5 and is told "There are 14 students, of whom 8 are female". At 11:ØØ A.M. these students leave and our lesson is no longer in active use. At some point room is needed for other active lessons (and commons), at which point our permanent common, with its numerical contents of 14 (students) and 8 (females) is sent back to permanent storage. At 3:ØØ P.M. the first student of the afternoon class causes TUTOR to prepare the lesson and fetch the permanent common from permanent storage without initializing the common variables to zero. The result is that she gets the message "There are 15 students, of whom 9 are female".*

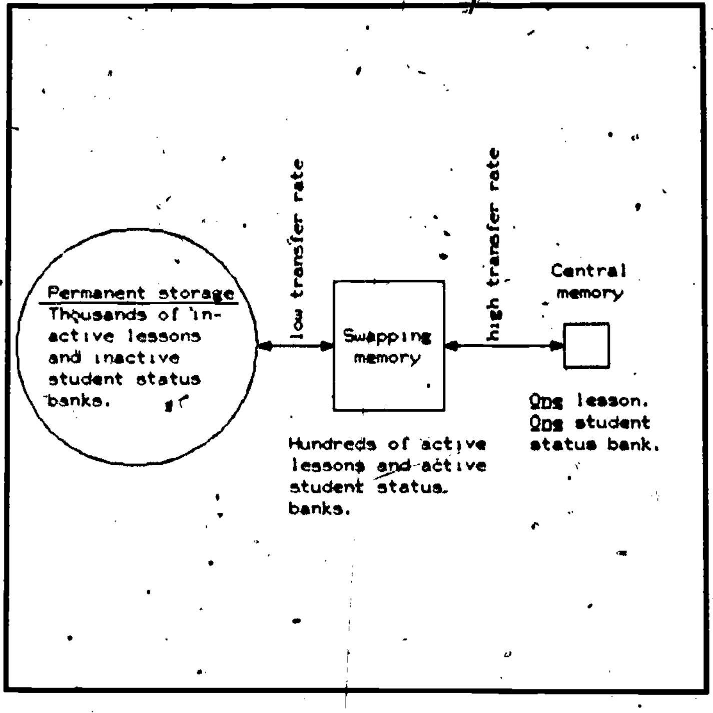
The key feature of permanent common is that it is fetched from storage when needed and returned in its altered state to permanent storage when the associated lesson is no longer active. In our case we could enter the lesson months after its initial use and see the total number of students who have entered the lesson during those months. Other uses of permanent common include the storage of data bases accessed by the students, such as population data in a sociology course or cumulative statistical data on student performance in the course.

The swapping process

Before discussing additional applications of common variables, it is useful to describe the "swapping" process by which a single computer can appear to interact with hundreds of students "simultaneously". The computer actually handles students one at a time but processes one student and shifts to another so rapidly that the students seem to be serviced simultaneously. In order to process a student, his lesson and his individual status (including the variables v1 through v15Ø) must be brought into the "central memory" of

* There is now an -initial- command which can be used to define a unit to be executed when the first student references the common. This makes it possible to perform initializations on a permanent common.

the computer. After a few thousandths of a second of processing, the student's modified status is transferred out of a central memory (to be used again later) and another student's lesson and status are transferred into central memory. This process of transferring back and forth is called "swapping", and the large storage area where the lessons and status banks are held is called the "swapping memory". The swapping memory must be large enough to hold all the status banks and lessons which are in active use; that is, in use by students presently working at terminals. It is not necessary for the swapping memory to hold in addition the many lessons not presently in use nor the status banks for the many students not using the computer at that time. These inactive lessons and status banks are kept in a still larger "permanent storage" area.


Permanent storage
Thousands of inactive lessons and inactive student status banks.
low transfer rate
Swapping memory
Hundreds of active lessons and active student status banks.
high transfer rate
Central memory
One lesson. One student status bank.

When a student sits down at a terminal and identifies herself as "Jane Jones" registered in "french2a", her status bank is fetched from permanent storage to see what lesson she was working on and where in the lesson she left off last time. If the lesson is already in the swapping memory (due to active use by other students), Jane Jones is simply connected up to that lesson, and, as she works through the lesson, her lesson and her changing status bank will be continually swapped to central memory. If, on the other hand, the required lesson is not presently in active use, it must be moved from permanent storage to the swapping memory. (This involves a translation of the TUTOR statements into a form which the computer can process later at high speed.) This fetching of the inactive lesson from permanent storage to prepare an active version in the swapping memory will typically be done once in a half-hour or more as the student moves from one lesson to another. In contrast, the swapping of the active lesson to central memory happens every few seconds as the student interacts with the lesson. Therefore, the swapping transfer rate must be very high whereas a low transfer rate between permanent storage and the swapping memory is adequate.

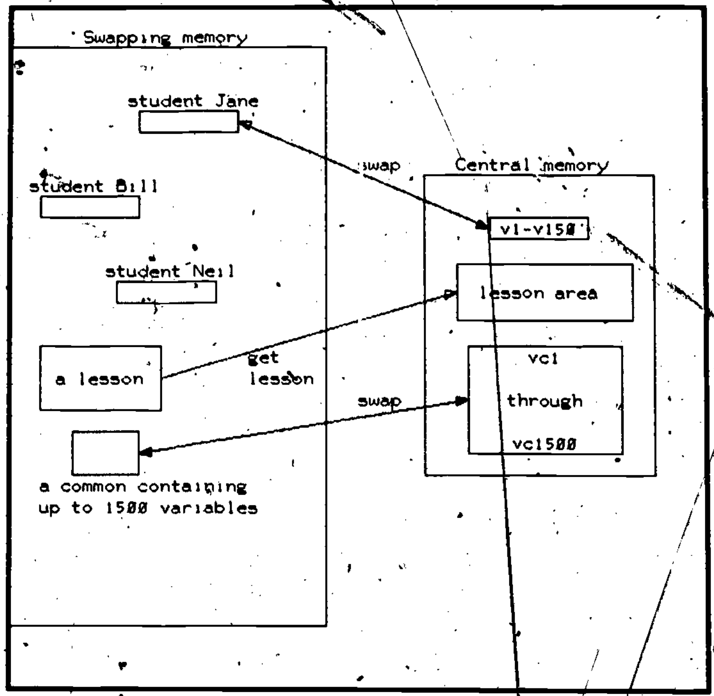
When the student leaves for the day, her status bank is transferred from the swapping memory to permanent storage. This makes is possible for her to come back the next day and restart where she left off.

The question arises as to why there are three different memories: central memory, swapping memory, and permanent storage. For example, why not keep everything in the central memory where students can be processed? It turns out that central memory is extremely expensive, but permanent storage in the form of rotating magnetic disks is very cheap. Why not do swapping directly between permanent storage and central memory? The rate at which lessons can be fetched from permanent storage is much too slow to keep the computer busy: the computer can handle only a small number of students because a lot of time is wasted waiting for one student to be swapped for another. If the cost of the computer were shared by a small number of students, the cost would be prohibitively high. In order to boost the productivity of the computer, a special swapping memory is used which permits rapid swapping. This minimizes unproductive waiting time and raises the number of students that can be handled. The swapping memory is cheaper than central memory but considerably more expensive than permanent storage.

There is, therefore, a hierarchy of memories forced on us by economic and technological constraints. The expensive, small central memory is the place where actual processing occurs, and there is never more than one student in the central memory. Material is swapped back and forth to a large medium-cost swapping memory whose most important feature is a very high transfer rate to central memory. Permanent storage is an even larger and cheaper medium for holding the entire set of lessons and student status banks. It has a low transfer rate to the swapping memory.

Common variables and the swapping process

Now it is possible to describe more precisely the effect of a -common- statement in a lesson. Just as an individual student's lesson and status bank (including the student variables v1 through v15Ø) are swapped between central memory and the swapping memory, so a set of common variables associated with the lesson is swapped between central memory and the swapping memory. There is in central memory an array of 15ØØ variables, called vc1 through vc15ØØ, into and out of which a set of common variables is swapped. As long as the -common- statement specifies a set of no more than 15ØØ common variables, this set will automatically swap into and out of the central memory array vc1 to vc15ØØ. (There is a -comload- command which can be used to specify which portions of a common to swap if the common contains more than the 15ØØ variables which will fit in central memory.) All 15ØØ variables in the central memory array are set to zero before bringing a lesson, status bank, and common into central memory, so that any of these variables not loaded by the common will be zero.



Note that the student status banks and commons are swapped in and out of central memory in order to retain any changes made during the processing in central memory. On the other hand, lessons are brought into central memory but are not sent back since no changes are made to the lesson. A lesson only has to be copied into but not out of central memory. The separation of the modifiable status banks and commons from the unchanging lessons makes it possible for a single copy of a lesson to serve many students.

It is dangerous to use iables without a -common- statement or to use vc-variables outside the e loaded by the common (e.g., referring to vc3 when there is a "common tatement in the lesson). For example, consider this sequence in a n which has no -common- statement:

```
        .
        .
        .
        calc    vc735<
        pause   2
        show    vc735
        .
        .
        .
```

This will show ∅, no ! The "pause 2" statement causes this student's material to be swapp t to the swapping memory for two seconds while many other students are essed. When he is swapped back into central memory, all the vc-variabl e zeroed. As a matter of fact, vc735 may temporarily take on many diff values during those two seconds as different students are processed. e other hand, a "common 8∅∅" would insure that vc1 through vc8∅∅ w e saved in the swapping memory and restored after two seconds, so th "18.34" stored in vc735 would again be available to be shown (unless been changed by a student using the same common who was processe g the two-second wait). Similarly, because the student variables v gh v15∅ are part of the swapped student status bank, the sequence

```
        .
        .
        .
        c        v126<=3.72
        p   e    2
        s        v126
        .
        .
        .
```

will correctly show "3.72". The contents of the student variables cannot get lost in the swapping process because these variables are saved in the swapping memory and restored to central memory the next time this student is processed.

The fact that common variables are shared by all students studying the lesson is extremely useful but can cause difficulties if you are not careful. Suppose you want to add up the square roots of the absolute values of vc1Ø1 through vc1ØØØ:

```
        .
        .
        .
        calc    total⇐Ø
        doto    8sum,index⇐1Ø1,1ØØØ
                total⇐total+[abs(vc(index))].5
        8sum
        show    total
        .
        .
        .
```

This iterative calculation will take longer than one "time-slice", the computing time TUTOR gives you before interrupting your processing to service other students. You are swapped out and will be swapped back into central memory later to continue the computation. It might take several time-slices to complete the computation, and in between your time-slices other students are processed. This time-slicing mechanism insures that no one student can monopolize the computer and deny service to others. Suppose two students, Jack and Jill, are studying this lesson and sharing its common. Suppose that Jack has reached the part of the lesson that contains the -doto- shown above. If, at the same time, Jill runs through calculations that modify vc1Ø1 through vc1ØØØ, her modifications will be made during the interruptions in Jack's processing. The total that Jack calculates will, therefore, be based on changing values and will <u>not</u> be the total at a particular instant. Jack calculates a partial total; Jill makes some changes; Jack continues in the -doto- to calculate some more; then Jill makes further changes, etc. At the end Jack has a peculiar total made up of partial totals made at different times. Even more drastic things will happen if "total" is itself a common variable: Jill might do "total⇐Ø" right in the middle of Jack's summation!

If it is necessary to get an accurate total at a specific instant, it is necessary to lock out Jill and other students from modifying common until the totaling is complete. The idea is to set a common variable to -1 before starting the calculation, then reset it to Ø upon completion. Whenever modifications are to be made, you check this "lock" to see that it is ∅ before making the modification. The structure looks like this, with vc1 used as the lock:

```
        .
        .
        .
       do        lock
      [   make         ]
      [modifications   ]
       do        unlock
        .
        .
        .
       unit      lock
       8chk      v15∅⇐vc1        $$ get present lock value
                 vc1⇐-1          $$ set lock if not already set
       branch    v15∅,x,8ok      $$ fall through if lock was already set
       pause     2               $$ wait two seconds
       branch    8chk            $$ try again
       8ok
       *
       unit      unlock
       calc      vc1⇐∅           $$ clear the lock
```

If vc1 is ∅, we set it to -1 and make the modifications, then reset it to ∅. If vc1 is -1, we superfluously set it again to -1 but fall through the -branch- to pause 2 seconds before again looking at the lock, in the hope that it has been cleared.

Note that the finish unit should clear the lock if the student who is modifying the common signs out before completing the modifications. Otherwise, the common will remain locked and other students will be hung up forever waiting for the lock to clear. One good way to do this is to keep track of which terminal ("station") locked the common: replace the statement "8ok" with "8ok   vc2⇐station". The system variable "station" gives an integer which is unique to the terminal. For a student at terminal 235, "station" has the value 235, whereas "station" has the value 472 for the student at terminal 472. The finish unit should then contain "calcs   vc2=station,vc1⇐∅,vc1" so that the lock is cleared only if this station was the one which had set the lock.

The use of v15∅ in unit "lock" is important. Here is a different version of that unit which will not work properly:

```
          unit   lock
          8chk   branch  vc1,x,8set      $$ check lock
          pause  2                       $$ wait two seconds
          branch 8chk                    $$ try again
          8set   vc1⇐-1                  $$ set lock
```

Suppose the lock is clear: vc1 is ∅. Then the branch is to label "8set" where we set the lock (vc1⇐-1) and proceed to modify the common. Unfortunately, we <u>might</u> get interrupted <u>during</u> the branch operation, in which case another student would <u>also</u> find vc1 still equal to ∅, set it, and proceed to modify common. When we get another time-slice after the interruption, we blithely set vc1 and proceed to modify common ourselves. Now we have <u>two</u> people modifying common, which is just what we were trying to avoid!

The version using v15∅ does not have this problem because TUTOR will not interrupt processing in the middle of a series of -calc- statements which do not involve -branch- or -doto-. This makes it possible to transfer the current value of the lock into v15∅ (v15∅⇐vc1) <u>and</u> set the lock (vc1⇐-1) without any danger of interruption between these two operations. On the other hand, TUTOR may interrupt on any -branch-, -goto-, -doto-, or similar branching command. The amount of the allowable time-slice used so far is checked on these branching operations in order to prevent infinite loops such as "8here branch 8here". This time check is performed on many TUTOR commands, especially commands which might require a lot of processing time. The only safe non-interruptable situation is within a non-branching -calc-. (In addition to time checks, TUTOR also makes checks for too much display material stacked up waiting to be sent to the terminal. TUTOR inserts a -catchup- command if necessary.)*

Note that a lock is needed only if different students are <u>storing</u> information into the same area of common. There is no problem with having different students <u>reading</u> information out of the same area of common or storing information in <u>different</u> areas of common. Logical conflicts arise only in <u>modifying</u> the <u>same</u> part of common. Even in this case there is no problem in many typical cases. In the example of counting the number of students in the lesson, we simply executed "vc1⇐vc1+1", which cannot cause any problem because all of the modifications are completed in one non-interruptable -calc-.

Instead of using "pause 2" to wait for the lock we could use "return". The -return- command has no tag. It means "return control to the computer--give me another time-slice as soon as possible after servicing other students". A -return- or -pause- can be placed just ahead of a small amount of branching computation to insure that this computation starts at the beginning of a time-slice. This insures a moderate amount of non-interrupted branching computation. For example, a -return- at the <u>beginning</u> of unit "lock" would make the version on this page work properly.

* There may be changes in these interrupt rules. It is strongly recommended <u>not</u> to use the lock techniques discussed above. Instead, use the new -reserve- and -release- commands.

The -storage- command

In certain applications 15∅ individual student variables are not sufficient, even using segmented variables. It is possible to set up extra storage of up 1∅∅∅ variables to give a total of 115∅ variables that are individual, not shared in a common. A "storage 35∅" statement will cause a storage block of 35∅ variables to be set up in the swapping memory for each student who enters the lesson. Like -common-, the -storage- command is not "executed"; it is rather an instruction to TUTOR to set up storage when the student enters the lesson. Like temporary common, the storage variables are zeroed when the storage is set up.

A -transfr- command can be used to move common or storage variables from swapping memory into the student variables or into the "vc" area. Usually, however, common is loaded automatically into the "vc" area. If the common is larger than 15∅∅ variables, a -comload- command must be used to specify which part of this large common is to be swapped into and out of which section of vc1 through vc15∅∅. In the case of -storage-, there is no automatic swapping: a -stoload- command is used to specify what parts of the storage are to be moved into what area of the "vc" variables. Here is a typical example:

```
common  1∅∅∅
storage 75
stoload vc1∅∅1,1,75
```

The common will be automatically swapped in and out of vc1 through vc1∅∅∅. The 75 storage variables will be swapped in and out of vc1∅∅1 through vc1∅75. It is good form to define all these matters:

```
define   comlong=1∅∅∅,stlong=75
         stbegin=vc(comlong+1)
         (etc.)
common   comlong
storage  stlong
stoload  stbegin,1,stlong
   .
   .
calc     stbegin⇐37.4
```

While -common- and -storage- are "non-executable" commands, -comload- and -stoload- are executable, so that swapping specifications can be changed.

The student's current variables v1 through v15∅ are saved with other restart information when he signs out. When he signs in the next day, these variables, therefore, have the values they had when he left. Storage variables are not saved, however. All storage variables are initialized to zero when the storage block is set up upon entry into the lesson, as with temporary common. If it is necessary to file away more than the standard 15Ø student variables, you could split up a common into different pieces for individual students. For example, if you need to save 2ØØ extra variables for no more than 2Ø students, you could split up a 4ØØØ-variable common into 2Ø pieces each containing 2ØØ variables. An alternative is "dataset" operations, currently experimental, which will permit you to control directly the transfer of blocks of individual data between the permanent storage (magnetic disks) and the swapping memory.

XI. Miscellany

This chapter will alert you to additional features of TUTOR and PLATO. Little detail is given. See appendix A for sources of additional information.

Other terminal capabilities

We have emphasized the keyboard and plasma display panel as the main input and output devices used in communicating with the student. Other devices which may be used include a projector of color photographs, a touch panel, a random-access audio playback device, and other specialized input-output devices. All of these terminal-associated devices are easily managed by TUTOR.

The plasma display panel is flat and transparent, which makes it possible to project photographs on the back, superimposing color photographs with plasma-panel text and line drawings. There exists a microfiche projector for the PLATO terminal which will project any of 256 color photos, with fractional-second access time to any of these 256 pictures. (A "microfiche" is a sheet of film carrying many tiny pictures.) Microfiches can be made from a set of ordinary 35-mm slides. Students or teachers can insert the appropriate microfiche in the terminal for the subject to be studied. The -slide- command selects any of the 256 photos: "slide 173" will project the 173rd photo. Additional options on the -slide- command permit the independent control of a shutter in the projector.

The touch panel is a device which puts a grid of 16 vertical and 16 horizontal infrared light beams just in front of the plasma panel. When a student points at the panel, he breaks a horizontal and vertical beam. The information as to which beams were broken is sent to the computer as a "key", and the lesson can use this information to move a cursor, choose a topic pointed at, etc. The system variable "key" contains the information:

```
        unit    getkey
        next    getkey
        enable
        pause
        goto    (key $ars$ 8),x,keyset,touch,extin,x
        write   Impossible!
        unit    keyset
        write   You pressed a key
                on the keyboard.
        *
        unit    touch
        calc    x⇐(key $ars$ 4) $mask$ o17
                y⇐(key $mask$ o17)
        write   You touched location
                x= ⊲s,x⊳ ,y= ⊲s,y⊳.
        *
        unit    extin
        write   The external input
                was ⊲s,key $mask$ o377⊳
```

The -enable- command permits touch inputs as well as inputs from any device connected to the external input connector at the back of the PLATO terminal. (The external input device might be a temperature sensor, an analog-to-digital converter, etc.) Without an -enable- command these inputs are ignored. A -disable- command will also cause inputs to be ignored. The system variable "key" contains a 1Ø-bit integer (see bit manipulations in chapter IX): the most significant or left-most two bits identify the source of the key (Ø for keyset, 1 for touch panel, 2 for external input), and the least significant or right-most eight bits contain the actual data (which keyset button, which touch panel beams, what external data). In the case of the touch panel, the eight data bits contain four bits of x and four bits of y to specify a position.

If an -enable- command is placed just after an -arrow-, touch inputs can be accepted. There is a -touch- judging command whose tag specifies a screen location and (optionally) a spatial tolerance: "touch location,tolerance". The -or- command is particularly useful here:

```
        .
        .
        .
        touch   1215
        or
        answer  book
        write   Yes, "libro" means book.
        .
        .
        .
```

The student will get the same message whether he types "book" or points at a picture of a book displayed at location 1215. (The -or- command can be used to make synonomous any judging commands. The system variable "anscnt" will be the same for all judging commands linked by -or-.)

There is a random-access audio device which stores twenty minutes of speech, music, or other sounds. Segments as short as one-third second can be accessed in a fraction of a second, no matter where the segment is located on the twenty-minute magnetic disk. As with microfiche, students can change the disks themselves. There is a -play- command to choose a section of the disk to play music or talk to the student.

Other devices can be connected to the external output connector at the back of the PLATO terminal and controlled with the -ext- command. The -ext- command can send up to sixty 16-bit quantities per second to a device. Among the interesting devices using this capability is a "music box" that plays four-part harmony!

Student response data

A crucial aspect of TUTOR on the PLATO system is that student response data can be collected easily to aid authors in improving lessons. Detailed information can be collected: unanticipated "wrong" responses (which may have been correct but inadequately judged), requests for help, words not found in a -vocabs-, etc. Summary information can also be collected: amount of time spent in an area of a lesson, number of errors made, number of help requests, etc. These detailed and summary data provide an objective basis for revising lessons.

A -dataon- command in a lesson turns on the automatic data collection machinery. Students registered in courses with associated response data files will have their responses logged in their data files. When registering students in a course, specific data collection options can be chosen. For example, one might collect only responses judged "no" (unanticipated incorrect responses). Anticipated correct responses (judged "ok") and anticipated incorrect responses (judged "wrong") would not be logged. This is often done because the anticipated responses are precisely those for which the lesson is already replying in a detailed, appropriate manner to the student. Here we see the difference between judge "no" (unanticipated) and judge "wrong" (anticipated). In this connection, -wrong-, -wrongv-, and -wrongu- make a "wrong" judgment, whereas the -no- command makes a "no" judgment.

The -area- command is used to subdivide a lesson into sections, each of which will produce an area summary in the data file. Each time the student encounters another -area- command, a summary of the previous area is placed in the data file. The area summary includes student name, area name, amount of time spent in the area, number of -arrow-s, number of ok/wrong/no responses, number of helps requested and found, etc. These summary data make possible a statistical treatment of lesson data which can pinpoint weak areas.

The -output- and -outputl- commands permit you to write your own information and messages into the datafile. This supplements the automatic data logging invoked with -dataon- and -area-.

While PLATO provides a standard mechanism for looking through data files (including sorting the data), you can also read back this information and process it yourself. For example, the -reada- command will read area summaries, and the -readl- command will read -outputl- information.

Routers and -jumpout-

A lesson can be designated to be a "router" which routes students through the many lessons making up a complete course. A router is associated with a course. Students registered in a course which uses a router will upon sign-in be sent first to the router, not to the lesson specified by the restart information. A typical router might ask the student, "Do you want to resume studying the lesson you last worked on?" If the student says yes, the router executes a "jumpout resume", which means "jumpout" of this lesson into the lesson mentioned in the tag, with "resume" having the special meaning "resume at the restart point". If the student says he does not want to resume, the router might offer the student an index of available lessons. Suppose the student chooses a lesson on the list whose name is "espnum". Then the router does a "jumpout espnum" to take the student to that lesson. (The -jumpout- command can be conditional.) Upon completion of lesson "espnum" (by "end lesson") the student is brought back into the router.* The router might then ask the student what he wants to do next, or the router might immediately take the student to an appropriate lesson.

Generally speaking, -jumpout- commands should be placed only in routers, not in instructional lessons. Following this practice insures that lessons can be plugged into routers on a modular basis. An exception exists in the case where one instructional package is spread over two or three physical lessons, in which case -jumpout- is used to connect them together.

A router can use up to fifty "router variables" (vr1 through vr5Ø) which are not affected by the instructional lessons. These can be used to keep track of which lessons have been completed, how many times they have been reviewed, how much time was spent in each lesson, etc.

Instructor mode

Authors write and test lessons, and students study lessons. Instructors choose lessons from the library of available lessons to make up a course for their students. Instructors also register students, monitor their progress, leave messages for the class or for individual students, etc. There is an "instructor mode" which makes it easy for instructors to do these things without knowing the TUTOR language. The instructor mode is based on a router together with a mechanism for setting up a roster of students. The options available through this router are sufficiently flexible to make it unnecessary in most cases to write specialized routers.

* If the lesson executed a -score- command, the router can use the corresponding value of system variable "lscore" to help decide how to route the student.

Special "terms"

Authors have a number of special "terms" to help them in curriculum development. If you press TERM and type "step", you can step through your lesson one command at a time. (A continued -calc- counts as one command.) This is enormously helpful in tracking down logical errors in a lesson. After each step, you can check the present value of student variables. At present you cannot step during the judging state. The judging state is completed and a switch made to the regular state before the step mode resumes.* There is also a -step- <u>command</u> which will throw the lesson into the step mode. The step features are operative only for authors testing their own lessons.

TERM-cursor provides you with a cursor which you can move around the screen using the "arrow" keys. Press f for fine grid or g for gross (coarse) grid. Also press f or g to update the display of the current cursor location. This facility is useful for deciding what changes to make in the positioning of displays on the screen.

TERM-consult notifies PLATO consultants of your request for help. When a consultant become available, he or she will talk to you by typing at the bottom of your screen. The consultant has on his screen the same display you have on your screen. It is as though the consultant were looking over your shoulder as you demonstrate the problem. You can talk to the consultant by typing sentences at -arrow-s or by hitting TERM and typing. (If you press NEXT, your sentence will be taken as a -term- to look for in the lesson. You can use ERASE to erase the line and type something else.) The consultants not only know TUTOR well but have a great deal of experience in helping authors.

TERM-talk asks you for the name of the person you want to talk to, then pages that person if the person is presently working at a PLATO terminal. The person called accepts the call by hitting TERM and typing "talk". The two of you can then talk to each other at the bottom of the screen, but neither of you can see what is on the rest of the other person's screen. If you want the other person to see all of your screen, press shift-LAB, which puts you into a mode similar to TERM-consult.

TERM-calc provides a convenient one-line desk calculator at the bottom of the screen. Authors get normal, octal, and alphanumeric results. To avoid confusion, students who use TERM-calc are not shown the octal and alphanumeric displays.

* This restriction has been removed. You can now step through judging commands.

APPENDICES

Appendix A

Where to get further information

The document "Summary of TUTOR Commands and System Variables" by Elaine Avner lists each TUTOR command and gives the basic form of the tag, and notes any restrictions such as maximum number of arguments or maximum length of names. Lesson "aids" available on PLATO provides detailed interactive descriptions of each command, as well as a wealth of other information useful to authors.

Lesson "notes" on PLATO provides a forum for discussing user problems. You can write notes to ask questions or to suggest new features that would be helpful in your work. You can read notes written by other users, including replies to your notes. Replies from consultants to programming questions generally appear within one day. (For faster service, use TERM-consult.) An extremely important section of "notes" is the list of announcements of new TUTOR features. Check this section regularly for announcements of new TUTOR capabilities. The announcements are followed within a few days by detailed descriptions in "aids".

Sometimes "notes" will announce a <u>change</u> in the TUTOR language involving an automatic conversion of existing lessons. For example, there used to be several different commands (line, liner, figure, and figuref) for doing what -draw- now does. When -draw- was implemented, all existing PLATO lessons were run through an automatic conversion routine to change the old commands into appropriate -draw- commands. It is probable that other such refinements will be made in the future. Therefore, be sure to

READ NOTES AND AIDS FOR CHANGES

that may have occurred since the publication of this book! The publication date on the title page of this book tells you where to start looking in the chronological listing of new features maintained in "notes".

It was indicated in this book that additional judging and graphics capabilities will probably be added to TUTOR. There is also work in progress to broaden greatly the handling of arrays in calculations to include matrix manipulations. Look for such things in "notes" and "aids".

Appendix B

List of TUTOR Commands

Display

at
write
writec
erase
eraseu
size
rotate
mode
charset
micro
char
plot
show
showa
showe
showo
showt
showz

draw
rdraw
circle
circleb
window
dot

origin
axes
bounds
frame
scalex
scaley
lscalex
lscaley
labelx
labely
markx
marky
locate
graph
gdraw
hbar
vbar
vector
polar
delta
funct

slide
play
ext

Calculations

calc
calca
calcs
define
do
exit
doto
goto
branch
transfr
zero

randu
setperm
randp
remove
modperm

pack
move
search
compute
itoa
clock
name
course
date
day
find
findall

common
comload
storage
stoload
return

Sequencing

unit
entry
nextnow
next,next1
back,back1
help,help1
data,data1
lab,lab1
term
base
end
restart

imain
finish

do
join
exit
goto
jump
jumpout

Student Responses

arrow,endarrow
long
jkey
copy,edit
force

answer,wrong
list
concept
vocabs,vocab
ansv,wrongv
ansu,wrongu
exact,exactc
touch
ok,no,ignore
ans
match
specs
or

storea
storen
store
storeu

judge
join

bump
put,putd,putv
loada

okword,noword

Other

pause
catchup
time
step
keytype
group

inhibit
enable
disable

dataon
area
output
outputl
reada
readl

Additional TUTOR commands not discussed in this book

abort	abort normal updating of common or student record
add1	add one to a variable
allow	allow an instructional lesson to use router common
altfont	use alternate font for all writing
ansva	character string match to student response
backgnd	run lesson at lower priority
change	change command names (e.g., to French or Russian)
chartst	check whether charset already loaded
close	like -loada- but takes one character per variable
dataoff	turn off student response data collection
dataop	like -helpop-
data1op	like -helpop-
delay	timed blank output for precise display timing
endings	specify word endings for -vocabs-
foregnd	run lesson at normal (non-background) priority
helpop	provide help on the page
helpop1	like -helpop-
iarrow	like -imain- but associated with -arrow-
iferror	specify unit to go to if -calc- error
kstop	like -back- but for the STOP key
labop	like -helpop-
lab1op	like -helpop-
open	like -storea- but stores one character per variable
press	presses a key for the student
readr	read a student record for data processing
readset	specify a data file for -reada- and -readl-
release	release a reserved common
record	record a message on audio device
reserve	reserve or lock a common
route	specify router units for end of instructional lessons
routvar	set up router variables
score	set a lesson score to be used by a router
sub1	subtract one from a variable
tabset	set up tabs for TAB key
use	use sections of another lesson to prepare this lesson

Appendix C

List of built-in -calc- functions

sin(x)	sine
cos(x)	cosine
arctan(x)	arctangent

Angles are measured in radians. For example, sin(45) means sine of 45 radians, but sine(45°) means sine of 45 degrees (0.707). The degree sign (MICRO-o) converts to radians. Similarly, arctan(1) is .785 radians, which can be converted to degrees by dividing by 1°, the number of radians in one degree; arctan(1)/1° is 45. Using the degree sign after a number is equivalent to multiplying the number by (2π/360). π (MICRO-p) is 3.14159....

sqrt(x)	square root; can also be written $x^{1/2}$ or $x^{.5}$
log(x)	logarithm, base 10
ln(x)	natural logarithm, base e
exp(x)	e^x
abs(x)	absolute value; abs(-7) is 7
round(x)	round to nearest integer; round(8.6) is 9
int(x)	integer part; int(8.6) is 8
frac(x)	fractional part; frac(8.6) is 0.6
=,≠,<,>,≤,≥	produce logical values (true=-1,false=0)
not(x)	inverts logical values (true↔false)
x and y	true if both x and y are true
x or y	true if either x or y is true (or both)
x cls y	circular left shift x by y bit positions
x ars y	arithmetic right shift x by y bit positions
x $mask$ y	sets bits where both x and y have bits set
x $union$ y	sets bits where either x or y has bits set (or both)
x $diff$ y	sets bits where x and y differ (exclusive union)
bitcnt(x)	counts bits

The operators involving equality (=, ≠, ≤ and ≥) consider two quantities to be equal if they differ by less than one part in 10^{11} (relative tolerance) or by an absolute difference of 10^{-9}. One consequence is that all numbers within 10^{-9} of zero are considered equal. Similarly, "int" and "frac" round their arguments by 10^{-9} so that int(3.999999999) is 4, not 3, and frac(3.999999999) is 0, not 1. This is done because a value of 3.999999999 is usually due to roundoff errors made by the computer in attempting to calculate a result of 4. The less than (<) and greater than (>) operators do not make these roundoff compensations.

System Variables

DISCUSSED IN THIS BOOK	NOT DISCUSSED IN THIS BOOK	
anscnt	baseu	aarea
args	capital	aarows
clock	dataon	ahelp
formok	entire	ahelpn
jcount	error	aok
key	errtype	aokist
opcnt	extra	asno
spell	judged	aterm
station	ldone	atermn
varcnt	lscore	atime
vocab	mainu	auno
where	mode	
wherex	nhelpop	
wherey	ntries	
	order	
	phrase	
	size	
	user	
	wcount	
	zreturn	

The third column consists of counters associated with the -area- command.