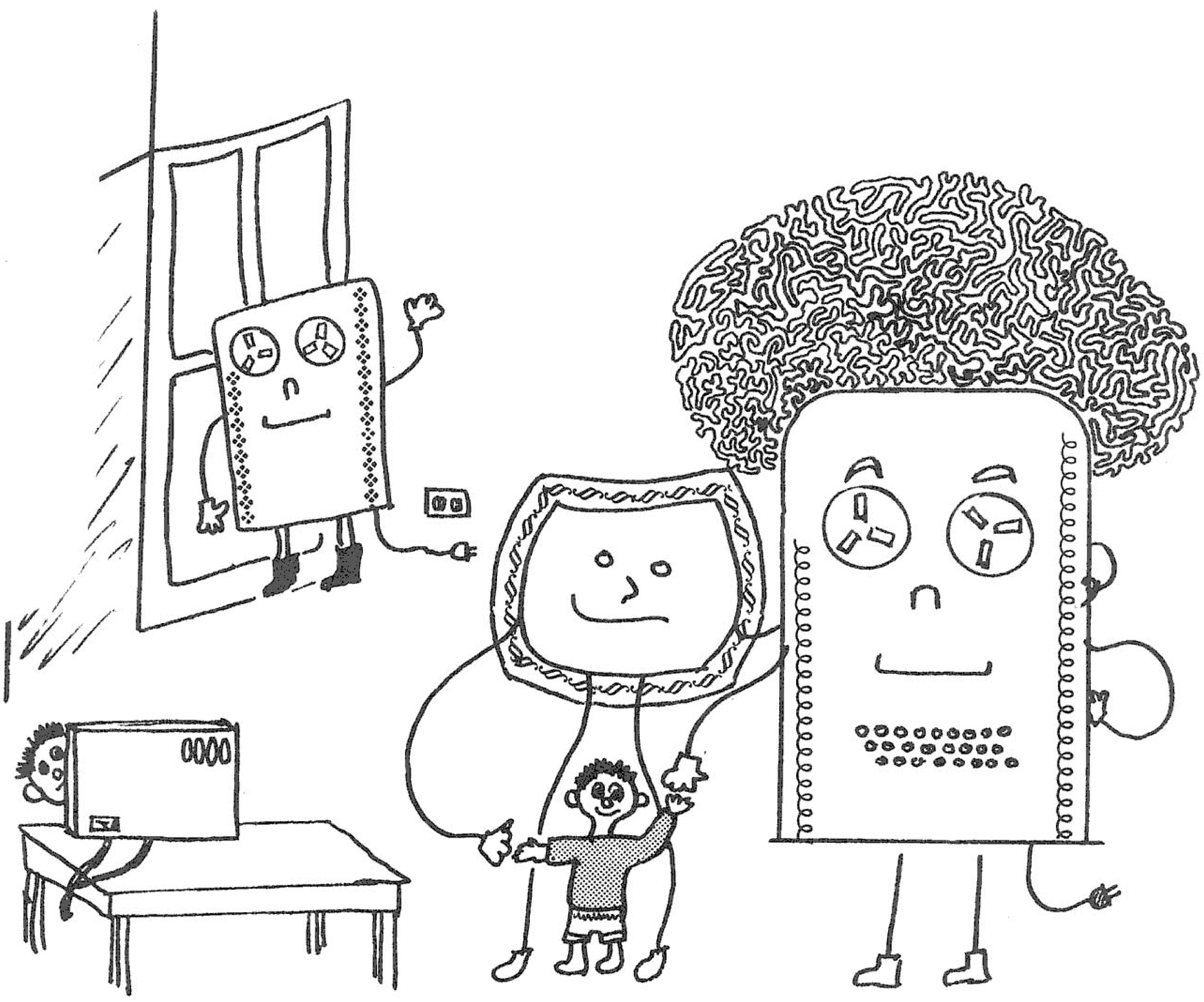
HYPERTEXT + TUTOR = HYPERTUTOR

The *Complete* Guide to

June 1 , 1974



YOU MAY OBTAIN MORE INFORMATION BY CALLING 312-492-5367 or writing to
COMPUTERS AND TEACHING, Northwestern University, 2003 Sheridan Road, Evanston, Il 60201

This document is also available on a microfiche.

The Complete Guide to HYPERTUTOR
copyright, 1974, Northwestern University, Computers And Teaching

INDEX

HYPERTUTOR 1
HYPERTEXT.....3
TURNING ON.....5
STUDENT MODE.....6
Entering Hypertutor 6
What happens "in" the lesson 7
Wacky lesson 8
Terminating lessons 8
The ONLINE system 8
HYPERAUTHOR MODE.....10
Daddy, where do lessons come from
Editor commands 11
More complicated editing 12
Deleting 14
Catastrophes 14
More information 14
ABOUT C.A.I.....15
Games 15
Simulations 15
Inquiry mode 15
Tutorial 16
Drill & Practice 16
Programmed Instruction 16
Further readings 16
HYPERCOMMANDS.....17
LEARNING ABOUT TUTOR.....18
Command and tag 18
The AT command 19
Write 20
Asking questions 20
Withholding the "ok" or "no" 22
Levels of response-checking 23
Branching 23
TUTOR Branching (table) 25
Calculating 25
Storage formats 25
Variable numbers 26
Assignment of values 26
Counting applications 27
Defining new variable names 28
Showing variable values 28
Numeric student responses 29
Storing character responses 29
Function keys 30

Erasing the screen 31
Inhibiting erasing 32
Conditionals 32
Special HYPERTUTOR variables 33
Indirect references 34
Graphics 34
Special characters 35
Writing sizes 36
WEIRDO TERMINALS.....38
DOCUMENTING IT.....39
Internal documentation 40
CONDUIT documentation guideline 41
SPECIAL "LESSONS".....43
Restoring "idled" lesson text 43
SAVE and RESTORE 45
Other interesting lessons 47
FFFAR-OUT IDEAS from
COMPUTERS AND TEACHING.....48
Computerized conferencing 49
A computer-based information exchange 50
Cottage industry 54
SPECIAL FEATURES.....55
Common storage 55
Random numbers 57
Jumpout 58
Vocabularies 58
Notes on programming technique 59
COST.....60
SYSTEM STRUCTURE.....64
Structure 64
Plato compatibility 64
Common storage, hyperauthor, access to otherprograms, lesson directory, condensing 65
Checkup, daily hyperinfo, "plato" version 66
The NUBB, Merlin's data, hypercommunications, files 67
COMMAND SUMMARY.....68
(Includes information on all existing HYPERTUTOR commands, in alphabetical order.)

Computer-Aided-Instruction (C.A.I.) has acquired a rather bad reputation in its short history -- perhaps connected with the generally poor attitude the "public" seems to have of computers in general. It has sometimes been billed as a panacea for education, providing individualized instruction and freeing the teacher for more creative tasks. It has also been extremely expensive, and seldom has the expense been justified by achievement of measurable educational objectives. C.A.I. is very "gimmicky" -- it is flashy, and schools are apt to buy equipment just to show that they are up-to-date, without realizing how much effort is required to program the computer and how much impact the new teaching tool will have on the kids and teachers.

The Computer Aids to Teaching project is attempting to formulate questions about these areas, to develop guidelines for programming and transfer of completed programs, and to help estimate the costs of C.A.I. in actual use at the university level. One of the outcomes of the C.A.T. project is the HYPERTUTOR, described in this booklet.

The HYPERTUTOR is a C.A.I. system which incorporates Theodor Nelson's HYPERTEXT idea (described later). A HYPERTEXT is a "text" which is not conveniently representable in writing, primarily because of its complexity and the desire that it be flexible (for use by student users). The HYPERTEXT can be manipulated by students and may be composed of C.A.I. "lessons", graphic presentations or other media, information retrieval packages, computerized conferences, statistical packages and other building blocks. Ideally, it must be connected to lots of learning resources outside of the C.A.I. system itself. The drawing on the page at the right illustrates the dilemma a user *should* face when approaching the C.A.I. system -- "*Wow! What should I learn about next?*"

HYPERTUTOR

An effective HYPERTEXT can only be built on a computer with lots of storage space for lessons and data. Thus, we emphasize the large-scale computer, with its great storage capacity and ability to process hundreds of students at one time. There certainly is no reason, however, that such a computer couldn't be linked to smaller *satellite* computers and terminals at educational centers, where specialized tasks would be performed.

My view of the C.A.I. future is of a large-scale computer, such as the PLATO-IV system, with thousands of users scattered across the countryside, perhaps working in *neighborhood centers*, where *learning facilitators* are available to help choose subjects to be studied and select resources to be utilized. Such a system need not be any more expensive than our current educational system -- much of the work could be done from the neighborhood centers or from homes. This guide illustrates (in a later section) that the cost of C.A.I. need not be as high as it was just a few years ago -- it is now within reach of most universities and many high schools.

HYPERTUTOR is a machine-language program which runs on CDC 6000 series computers. It requires little central memory (less than 20000 words), depending upon memory necessary for a particular lesson and upon options selected by the installation, and very little central processor time

(2.5 to 5.0 milliseconds of CPU time per student-second on a CDC 6400). And it accepts the TUTOR-IV programming language, which is compatible with PLATO-IV. Programs from PLATO can therefore be fairly easily moved to HYPERTUTOR, and HYPERTUTOR lessons are guaranteed to run on a PLATO system.

A current problem with PLATO is the high cost of terminals (over $5500 each). HYPERTUTOR will drive any sequential line-by-line terminal a computer system will drive. In our Northwestern system, we also drive Hazeltine-2000, ADDS 580 and 880 terminals, using the full range of graphics available. The PLATO-IV terminal may be driven from any ASCII-sending computer, through a device called the NUBB, which requires only the special programming of the "front-end" computer, since PLATO-IV terminals are full-duplex with some additional complicating characteristics. HYPERTUTOR supports full upper/lower case, line drawing, character definition and plotting, touch-panel, audio-unit, slides and external channel equipment.

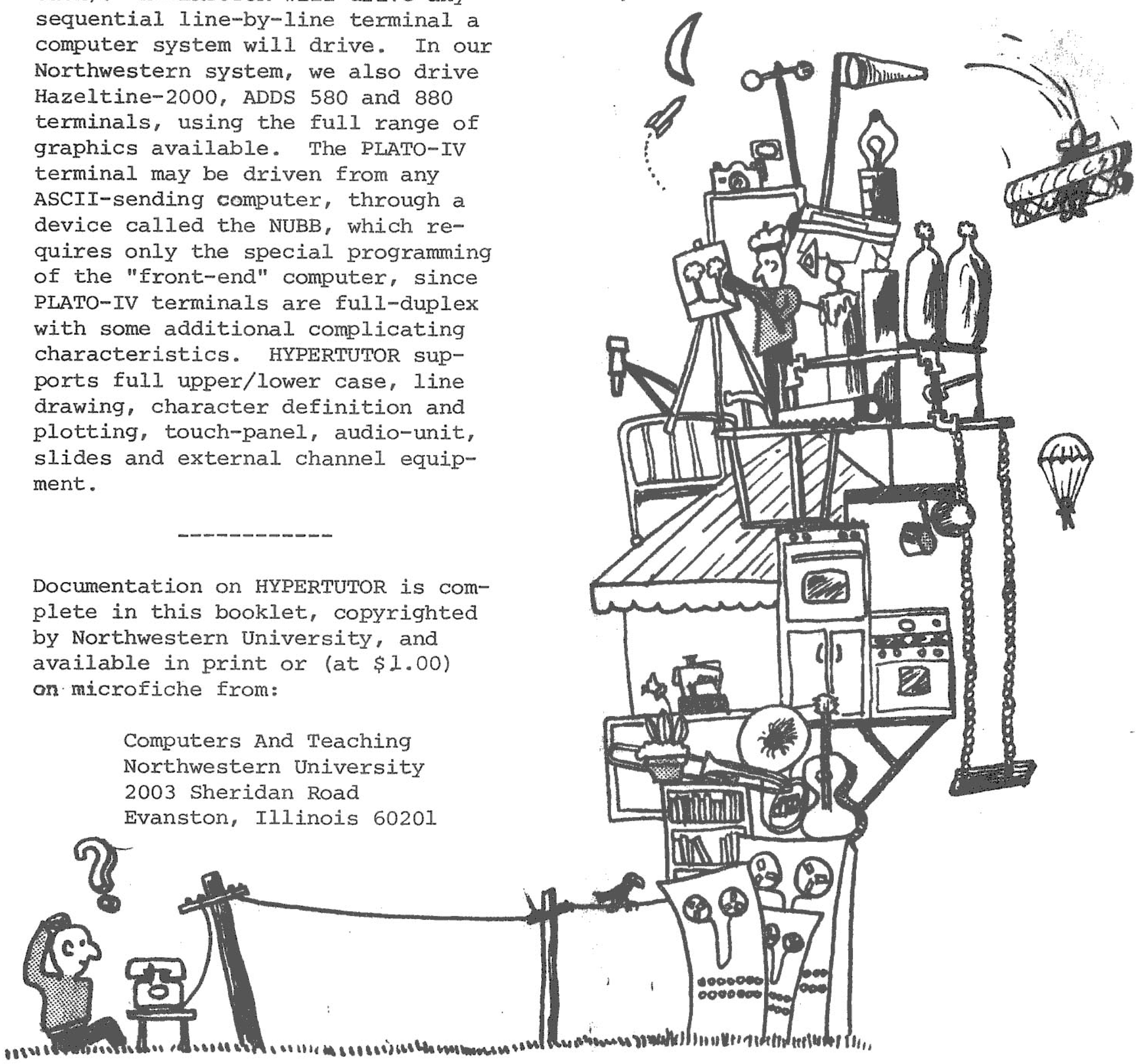
Documentation on HYPERTUTOR is complete in this booklet, copyrighted by Northwestern University, and available in print or (at $1.00) on microfiche from:

Computers And Teaching
Northwestern University
2003 Sheridan Road
Evanston, Illinois 60201

The computerized C.A.I. Learning Exchange -- "What shall I learn about next?" -- provides telephone access to a computerized bank of learning resources. This is an ideal we hope to make a reality in the near future!

Jim Schuyler

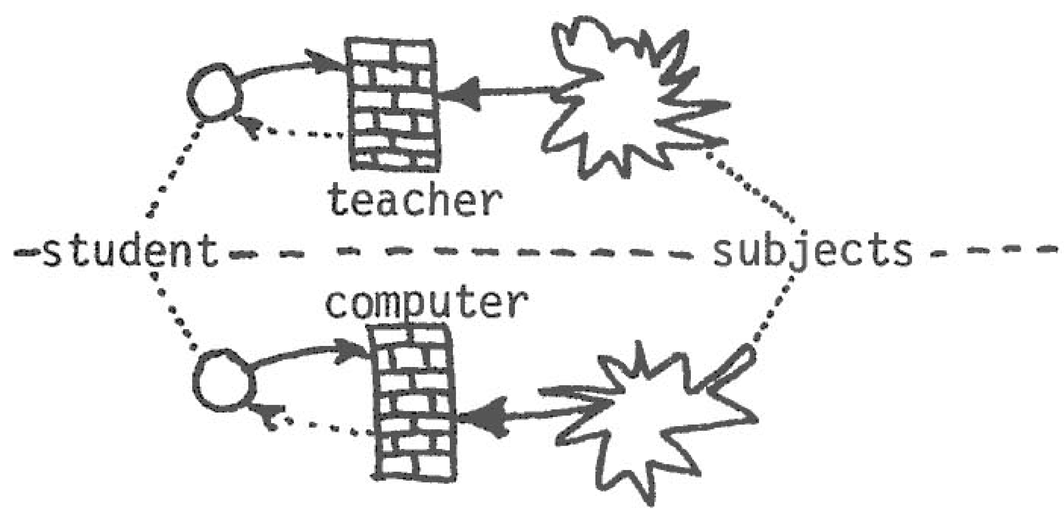
Jim Schuyler, Director



③

HYPERTEXT

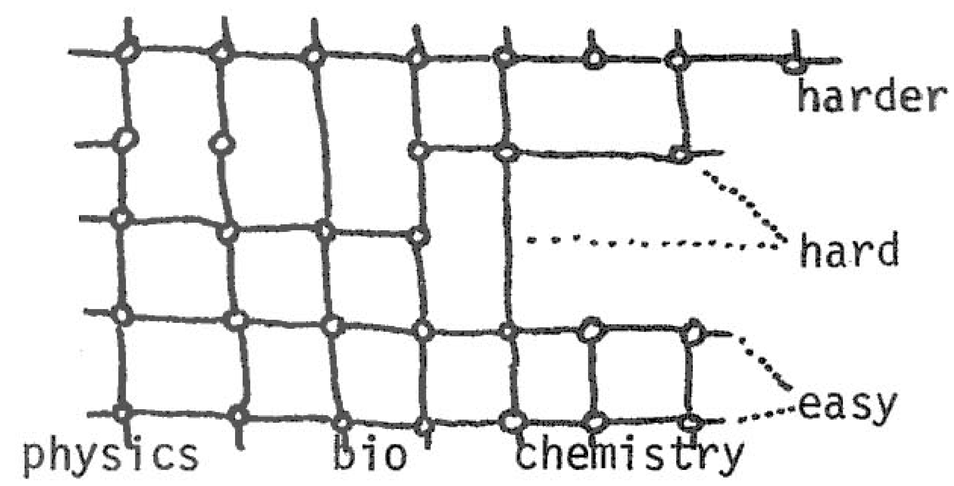
HYPERTEXT is a word coined by Theodor H. Nelson to describe his approach to Computer Aided Learning. A HYPERTEXT exceeds the bounds of conventional C.A.I. by putting the student in the driver's seat, contrary to normal practice...



According to Nelson, the trouble with C.A.I. is that it's too much like what we're trying to get away from -- it doesn't let the student freely explore the things which interest him. Instead it presents the "official" angle and its officially presented tie-ins. Both the regular classroom and CAI remove opportunities for initiative; they both bore the student, they give grades and narrow the curricula in order to produce "results".

In an ideal instructional system (a HYPER medium) "enthusiasm and involvement are what really count". The student is given an environment to explore, and is motivated only enough to get him started. Discrete (chunk style) HYPERTEXTS consist of separate pieces of text connected by links. Ordinary prose may appear on the screen and be moved forward or backward by throttle. An asterisk may indicate a connection with some other point in the hypertext (a connection found and tagged by the student usually). A HYPERGRAM or performing picture might explain how a motor works or show how the sine and cosine vary as an angle changes. A map might be projected, and the student may move it around, blow it up or make it smaller, at will. He might request additional overlays, such as population and climate. Or there might be queriable illustrations -- the student points at part of the picture and gets an explanation of its function.

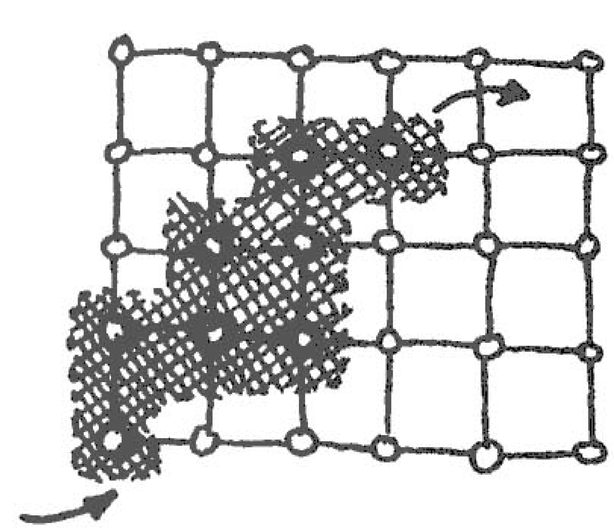
In HYPERTEXT, the student is in control. We could imagine a two-dimensional hypertext as a network of points with links between them (some links perhaps jumping around or between nodes).



In such a hypertext the student might "fly" above the lattice, looking at descriptions entered by authors of the lessons at various nodes (or at comments by students who had taken those lessons) and might then decide to "take" a particular lesson. If the material turned out to be too difficult, he would terminate the lesson and start "flying" to a "lower" node, where material presented would be less difficult. The columns of the network might contain related types of materials, varying in difficulty.

Other variations are, of course, possible!

Note that not only could the student change position in the network of lessons, but the lessons themselves could instruct the computer to switch lessons. This makes it possible for a student to enter a relatively simple lesson, learn more and be switched to progressively more difficult materials, and finally exit the lesson on a completely different level.



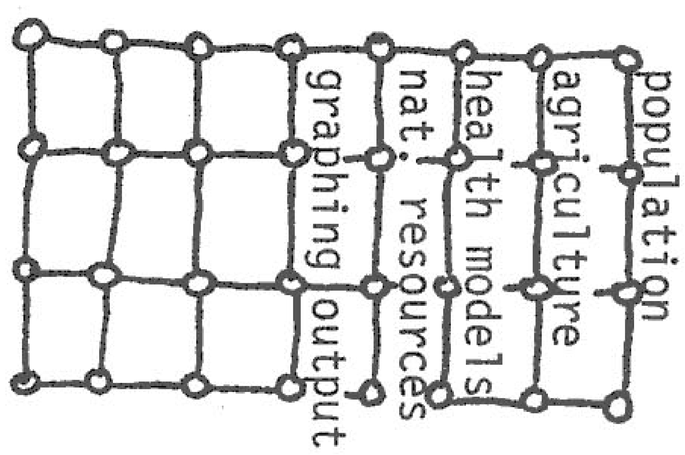
Of course, a HYPERTEXT doesn't have to be two-dimensional -- it's just easier to visualize it that way. As one finds more ways of describing educational materials, more dimensions may be added.

A personal dimension for instance -- in which the student may make personal connections between nodes in the network. All connections are remembered by the computer and displayed upon request, so the student can trace through previously recorded connections.

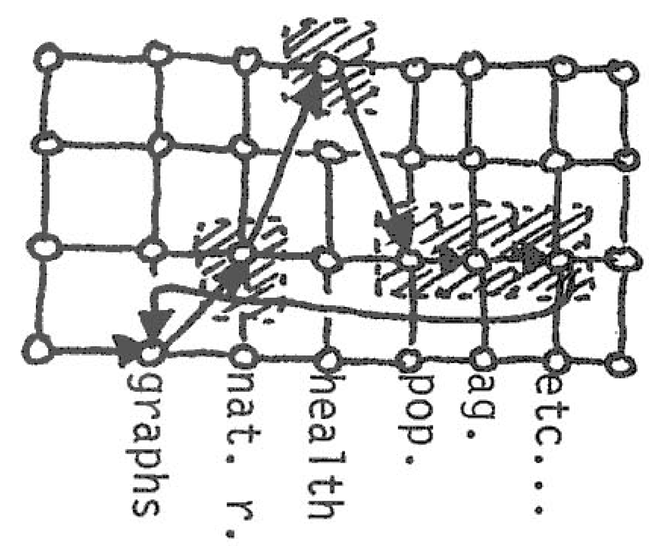
A "teacher" could create a small space within the larger HYPERspace, in which a group of students could work on a particular problem.

Dick Walz and I have thought about the problems of simulation in such a system as HYPERTEXT, and have created this model to guide further experimentation:

A simulation model (of the world, let's say) is created of a number of discrete parts, each of which is entered as a node in a section of the HYPERTEXT. A world model would be composed of population projections, food supply, natural resources, disease, etc. The student is given a choice of several pre-packaged models in each field, shown below.



Once the student has made a choice of model for each sub-section of the world model, a set of general input and output nodes is also chosen (to graph outputs, for instance). Then, the master-control node sets up system parameters and jumps to the first of the selected models. Each model computes the next projection (next time-period) and then jumps to the model in the next sector selected by the student. When all of the nodes have finished computing, the graphic output node is entered and a graph is produced on the student's screen. The student modifies assumptions, makes a different selection of models, and tries again until the desired effect is achieved.



What is unique about such a model? First of all, it provides the student with the chance to make a choice among many possible models. This is actually what a decision-maker or planner must do every day. Second, once the model has been chosen, it is possible for the student to revise that choice until a reasonable outcome is achieved. And finally, because the model is implemented within the HYPERTEXT, each node is accessible to programmers and may be easily changed or adjusted without throwing the whole model out of whack.

The various sectors of the model have access to each other's data, and to results of previous computations. In a conventional C.A.I. or other programming system it would be terribly difficult to establish such linkages among independent programs!

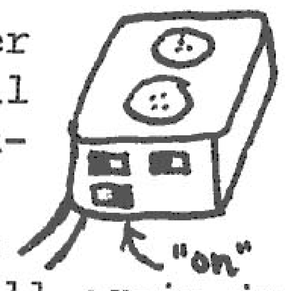
5

TURNING ON

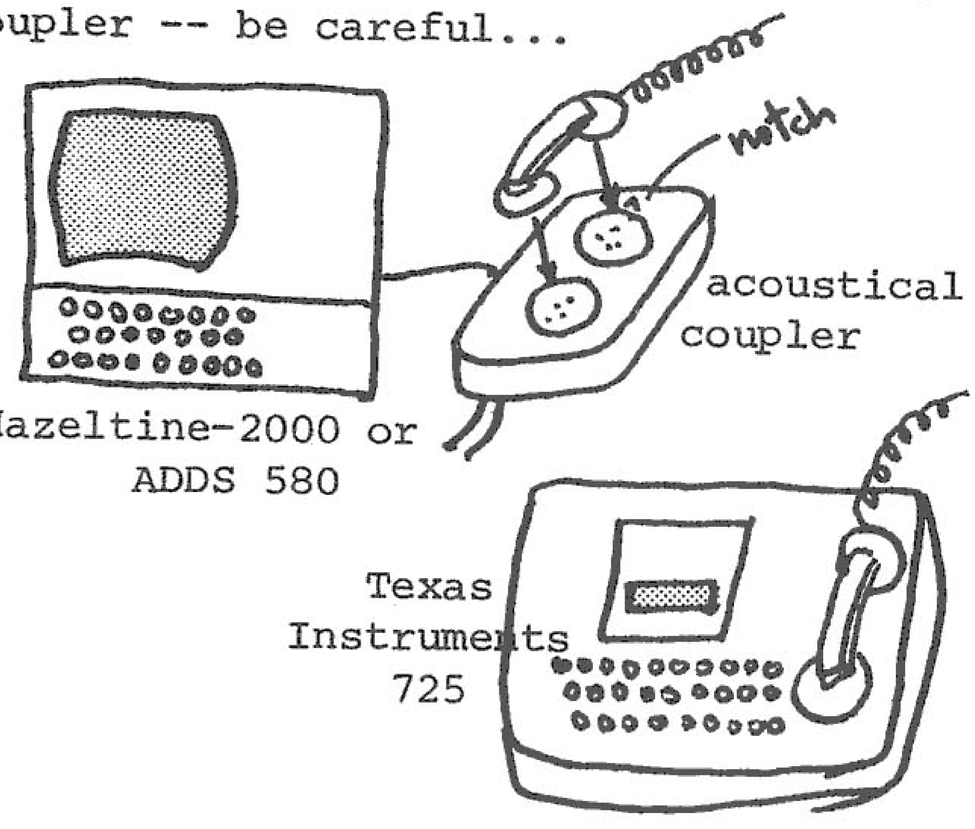
Here's how to turn your terminal on to HYPERTUTOR!

1) Plug in the terminal and turn it on. If the terminal has both LINE (ON-LINE) and LOCAL modes, use LINE. Be sure the acoustical coupler (below) is on too.

2) We recommend 30 character per second operation -- call 7110 (312-492-7110 from outside) on the phone by the terminal. If the number is busy or does not answer, call again in a minute or two.



When the computer answers (with a tone) you place the telephone handset into the cradle of the acoustical coupler. Don't tarry, the computer will hang up if you're too slow! The phone cord is supposed to go in the notched end of a coupler -- be careful...



3) Press "return", "cr" or "new line" to see if the computer is communicating. It will type an error message and say TERMINAL IDLE if it's working. It is always wise to try this step. If there is no response, you may wish to try calling up again later.

4) Type your charge number. It is composed of two letters, four numbers, a dash and four more numbers. You must get it exactly right! If you blow it, you must press "return" and start all over. Most class charge numbers also require that you place another dash, followed by your initials or name, after the charge number. Example...

charge number

XX0000-0000-JIMMY

You must press "return", "cr" or "new line" after every line you type. Be sure you press it at the end of your name.

5) If your charge number is valid, the computer will ask for CONTROL CARDS. Your control card is...

%$CAI

It stands for Computer Aided Instruction, and you must type it exactly as indicated. Be sure you press "return", "cr" or "new line" after typing in each line.

The computer will ask you your name and the type of computer terminal you're using. Type the model number (this will normally be "2000", "580", "880","725" or "33". A system message will be printed -- it often contains important information about HYPERTUTOR -- please read it.

6) After the message LOADING HYPERTUTOR is displayed, you are under control of the HYPERTUTOR system. It will tell you what options you have -- currently they are:

1) to type a lesson name,
2) to type "browse", or
3) to type "help".

Most students are told the name of a particular lesson they are to run -- yours is

lesson name

If you can't figure out what to do at this point, type HELP. You can always leave a HYPERTUTOR lesson by typing the characters

#END

You are then returned to step 6, above.

7) To terminate the computer session, type

%LOGOUT

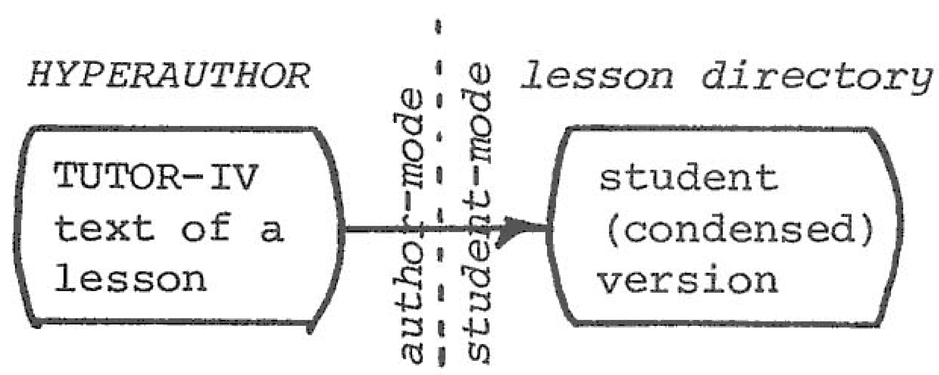
and hang up the telephone. Be sure you turn the equipment off before you leave.

Computers And Teaching	*312 492*	*5367*
Computer operator		*3695*
C.A.T. terminal room		*5517*
Computing Center		*3682*

HANDY PHONE NUMBERS

STUDENT MODE

Most HYPERTUTOR users use *student mode* almost exclusively. This is the mode of operation in which one browses thru the hypertext to select a lesson, and then enters (or "takes") the lesson. It can be contrasted with the *author mode*, in which a user creates or modifies lessons to be used by students. This section of the GUIDE is intended as a roadmap to help you understand how the student mode functions, how to use it to select lessons, and how it relates to the author-mode.



The diagram above illustrates the relationship between what an author does and what is available to the student. Lessons are generally written in a language called TUTOR-IV. These are stored and modified as desired to produce a fully completed *lesson*. The computer takes the text and compresses (or condenses) it into a special version which is put into the lesson directory and can then be used by students. It is always the condensed version of a lesson which is used to present information to students on-line.

The lesson directory is updated each time the author changes the TUTOR-IV text for that lesson. The lesson directory also contains a special list of *public* lessons, which is updated daily.

ENTERING HYPERTUTOR

The TURNING ON section of this guide explains how to get a computer terminal connected to the computer. Once the user has typed %$CAI (step #5), s/he is under control of HYPERTUTOR and can only terminate the computer session by typing %LOGOUT, or by hanging up the telephone. The initial display mentioned in step #6 of that page looks like this:

```
---HYPERTUTOR---
Type the name of the lesson you want, or
type "browse" to see some lesson names,
or type "help" to learn about HYPERTUTOR.
                    >

(type %LOGOUT before hanging up)
```

The display will be changed from time to time (as we find clearer ways of saying what we mean, and as new options are added). It is important that you read this page and not "mechanically" just type "browse" or a lesson name, since it will be adjusted without notice.

When special lessons are prepared by a teacher for use by a single class or group of students, they are normally classified as *private* lessons. The name of a private lesson is never shown on the screen, and therefore it <u>must</u> be requested while the student has this initial display on the screen. If the student knows a lesson name, <u>any</u> lesson may be requested by typing its exact name at the arrow (>). Thus, students who know the name of a public lesson may access it without going thru the browsing mode at all.

HYPERTUTOR always displays an arrow (>) to indicate that it is your turn to type. Normally, one types a response to a question and <u>always</u> presses the NEXT (or carriage - return)key to signal that s/he is ready to go on. The computer will always wait for the NEXT key before proceeding. Special function keys (like HELP) are discussed later for those using the PLATO terminal.

If, at the initial display page, the student types HELP (or presses the HELP key on a PLATO terminal), a special lesson is called. It introduces the concept of a *hypertext* and explains the *browsing mode*.

If the student selects browsing , a second page (called the browsing page) is displayed:

```
 --browsing--  --hypertutor--
type "more" for information on a lesson,
or type "help" to find out how to browse

              jotto
         a challenging word game


          >
```

An initial lesson name is always displayed in the center of the browsing page (here it is "jotto") Normally it will be the last lesson used by the student. To randomly move from one lesson to another the user presses the NEXT key (we will always call the "carriage-return" key the NEXT key, because that is what it is called on the PLATO terminal and in the TUTOR language). Whenever NEXT is pressed, the name on the screen is replaced by a new name and title. As on the initial entry page the student may type the name of any lesson to run it, even though the lesson's name doesn't appear on the screen at that time.

For the student to make judgements about lessons, s/he needs more information. A student may type the word "more" to obtain more information about any lesson whose name and title are displayed on the screen. Usually there will be an author's name and phone number, with a short subject-matter description, approximate level of difficulty, completion status and sometimes some *links* to other lessons. The four most common links are called UP, DOWN, RIGHT and LEFT. A sample of what appears on the screen appears below:

```
 --browsing-- --hypertutor--
type "more" for information on a lesson,
or type "help" to find out how to browse

              jotto
         a challenging word game


          >more
subject a word/spelling game in which
        you guess the computer's secret
        word.
author  lew nathan
level   high school or older
up      password
left    gametheory
right   guesser
```

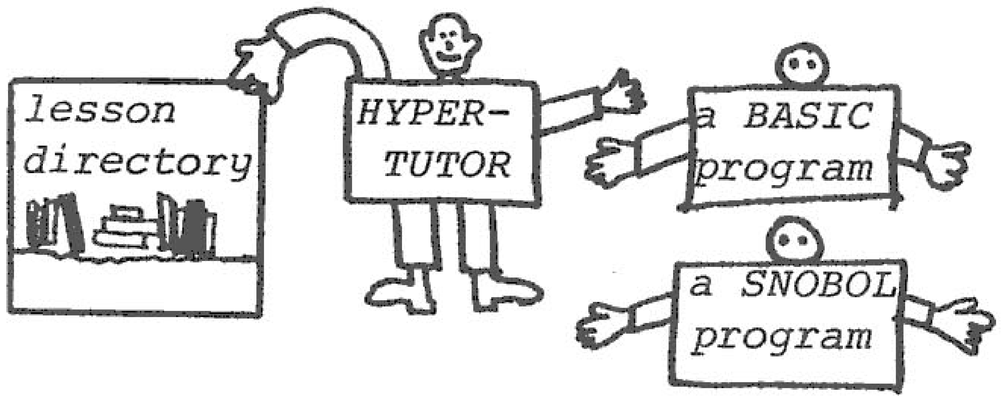
In this case, three special links have been specified. UP links usually lead to a harder lesson, DOWN links to an easier lesson. RIGHT and LEFT lead to related lessons in different subject areas. The author of a lesson is free to name his links anything he wishes, and often you will find them named *harder* or *easier*. If the student wishes to pursue one of the links, s/he types the name of the link (UP, for example) -- the display will then be changed so the additional lesson is shown. The student may request "more" again, to search for further links.

Even while browsing, the student may always request a lesson directly by typing its name.

WHAT HAPPENS "IN" THE LESSON

After you type a lesson name, HYPERTUTOR clears the screen and prints the current time and date at its top. (This is also true when a lesson voluntarily transfers control to another lesson.) All of the displays which then follow are created by HYPERTUTOR as it interprets the instructions given by the author of the lesson you've chosen. The text of the

lesson is physically located in the *lesson directory* all of this time, and the HYPERTUTOR simply examines this text to determine what to display on the screen and how to interpret your answers. Because HYPERTUTOR works in this fashion, it is called an *interpreter*. The diagram below illustrates how it works:



HYPERTUTOR is an active part of the system, while the directory is not. The HYPERTUTOR "grabs" information from the lesson directory according to your choice of lesson (like reading a cookbook) -- but when it needs programs in other programming languages (like BASIC or SNOBOL) it relinquishes control to those programs. It regains control when the other program returns control to it, or when the user types *%DROP*.

WACKY LESSONS

Novice computer users sometimes can't believe that something the computer "says" can be wrong! If you get to a point in a HYPERTUTOR lesson where you can't make head or tails of what's on the screen, or can't seem to get the right answer to a question (or you know you're right and the computer's wrong), just chalk it up to experience -- the author of the lesson probably blew it in some way. Perhaps the best strategy is to note the error, being very explicit about what was on the screen and what you were answering to the question, and then to forward your note to the author of the lesson. You can do this through the C.A.T. project, if you wish.

TERMINATING LESSONS

In many cases a student will legitimately wish to terminate a lesson before the author gives him a chance to do so. In such cases, we recommend that the user type *#STOP* (no period on the end!) to terminate the lesson. Users on the PLATO terminal may depress the SHIFT key and press the special black key marked STOP. The user is returned to the initial entry display and may chooose to browse, or may run another lesson directly.

Since the HYPERTUTOR can link to lessons in other programming languages, we often recommend that the user type *%DROP* if the procedure indicated above does not immediately terminate a lesson. %DROP will *always* terminate programs (except for the EDITOR) and return to HYPERTUTOR!

THE ONLINE SYSTEM

Most HYPERTUTOR users will have little opportunity to learn about commands to be given directly to the ONLINE system, which controls the sending and receiving of characters to all on-line terminals. ONLINE handles the sign-on procedure explained in the TURNING ON section, accepting your charge-number and asking for the control-card %$CAI, which gets you into HYPERTUTOR. The few commands you may need to know are listed here, with brief explanations:

%HOLD
This command will cause the computer to suspend the operation of your job for up to two hours. You can later continue, perhaps from another location or on another type of computer terminal, using %HOOKUP.

%HOOKUP-jobname
This command is used to connect your terminal to a previously held job (or to one lost due to computer failure or an accidental telephone disconnection). Always put the *jobname* after the dash -- exactly as it was typed when you started it, or when you held the job.

%M
If you type %M followed by a one-line message, the computer will print it on the operator's console. He can respond to these when mounting tapes or when your job is in "trouble".

It is the ONLINE system which causes the computer to process the student's *job*. Because there may be many of these jobs active (many students interacting with the computer at the same time) the computer must *schedule* the operation of these jobs in such a way that everyone is kept happy. Keeping people happy is usually measured in terms of low *response-time*. The response-time is the time it takes the computer to respond to you after you type something and press NEXT. On the CDC 6400 at Northwestern, you'll find that response time is usually about a second (or less), but increases to perhaps a few seconds in the middle of the afternoon, when there are more users demanding action. HYPERTUTOR's response time will vary from computer to computer, because of the scheduling problems, which are complex.

The CDC 6400 computer is composed of a *central processor, ten peripheral processors, a central memory, an extended core memory* and a number of *disk* storage devices. Also available for slower retrieval of information are many *magnetic tapes*.

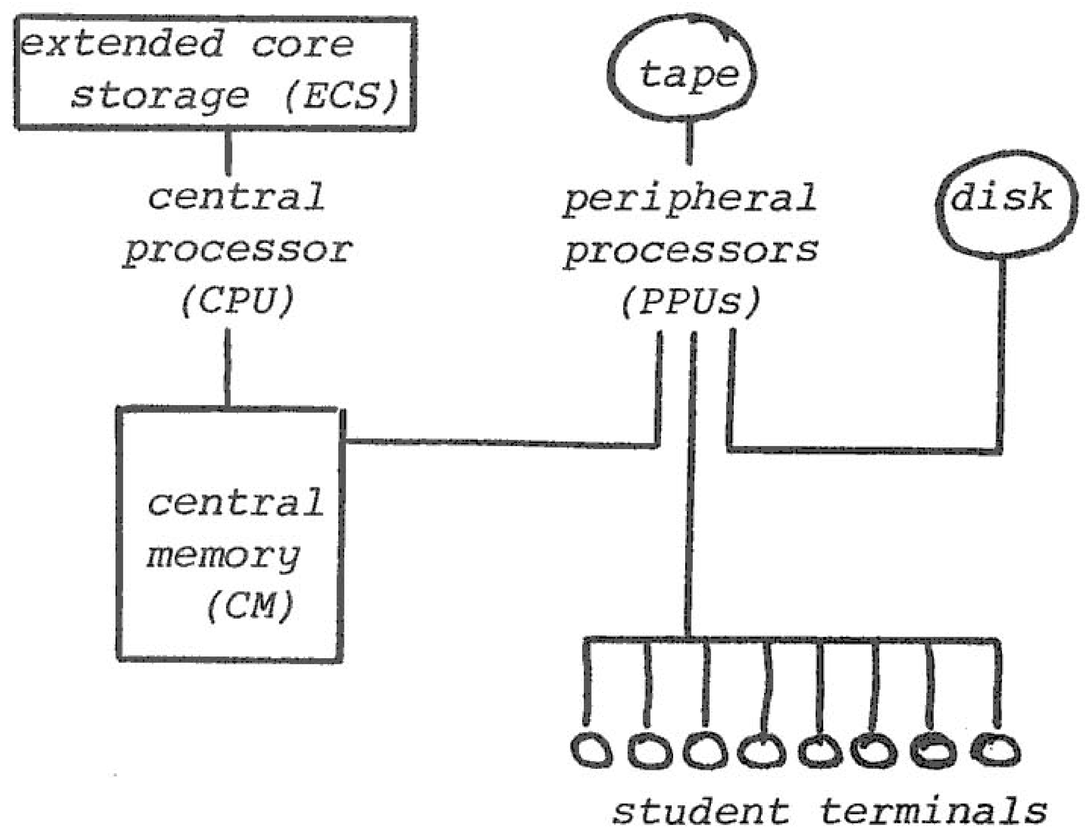
The *central processor* is sometimes called the "head" by computer builders, because it does the brainwork. The central processor (CPU) calculates the correct answers to questions, determines which unit is to be processed next, calculates values, creates the displays, and generally directs the show when HYPERTUTOR is running. It is the fastest part of the computer.

Peripheral processors are slower than the CPU, and perform less flashy tasks like reading magnetic tapes, finding and reading or writing information on the disk storage devices, and carrying on communications with the student's terminal. They always take directions from the CPU (as viewed from a programmer's standpoint). However, they are the beasties who schedule the operation of ONLINE, and are very important to all of us. They are run by a rather large library of *system programs* written by people called *systems programmers*.

The *central memory* is where data and programs are kept for the central processor. HYPERTUTOR is always in central memory when it is calculating. When HYPERTUTOR is waiting for you to respond to a question, however, the peripheral processors copy the portion of central memory it would normally occupy onto a disk unit, to release the memory for use by other students. When you finally type your answer and press NEXT, HYPERTUTOR is read back into central memory and begins calculating the appropriate action again. This process of swapping your job in and out of central memory is part of what is called *time-sharing*. The computer shares its time among many users!

Extended core memory (or storage), ECS, can be used to hold jobs swapped out of central memory, or can be used for holding data. It is much larger than central memory, but is harder to access and is slower.

Disks are very large storage devices, holding about 100 million characters each, compared to the 650,000 characters the central memory could hold. Northwestern has about 370 million characters storage capacity on disk at the present time. They are slower than central memory, taking about 20 thousandths of a second to access, compared to about 1 millionth of a second for central memory. HYPERTUTOR lessons are kept on disk units unless "idled" due to inactivity.



HYPERAUTHOR MODE

DADDY, WHERE DO LESSONS COME FROM?

After one has been introduced to the HYPERTUTOR, it is reasonable to ask "How are lessons created?" Paradoxically, each and every lesson is created by another lesson: HYPERAUTHOR. It lets an author create TUTOR-IV files, of which condensed copies are made and inserted automatically into the *lesson directory*. (See the student-mode explanation.)

HYPERAUTHOR is "entered" like any other lesson -- by typing its name on either the initial entry page or the browsing mode page. It is a *private lesson*, and will not appear while students are browsing. It allows authors access to an editing program, called EDITOR, in order to create and manipulate files of text with ease.

***

Each institution which uses HYPERTUTOR utilizes its local EDITOR for creation and modification of TUTOR-IV text files. This is done so that users who already know how to use the local editor need not retrain to use HYPERAUTHOR. And those encountering an editor for the first time will gain a skill valuable to them in other contexts on their computer system. *All of the information to follow is applicable only to Northwestern's EDITOR!*

***

The HYPERAUTHOR manipulates text stored on disk units (see previous page), and special directions must be followed for text received on tape or cards.

When one enters HYPERAUTHOR, the EDITOR immediately prints:

EDITOR READY
AUTHOR MODE
WHAT LESSON?

and waits for a response.

At this point the author must type the name of a lesson to be created or modified (we will begin by assuming in the example that the lesson is new). Once a lesson has been named, it is difficult to change that name, and lesson names should therefore be picked with care. They may contain from one to seven alphabetic (A to Z) characters -- no "special" characters, such as commas, parentheses, dashes, etc. are allowed. The author types the lesson name and presses NEXT (or "carriage return", "new line", "return", remember?). The HYPERAUTHOR next asks:

WHAT CHANGE CODE?

The *change-code* is a one to seven character word (which may contain numbers) used to determine whether a person is authorized to access the TUTOR text of a lesson. Because this is a new lesson, you may pick the change-code freely -- if it were a previously created lesson, you would have to know the exact change-code specified by its author in order to even see the TUTOR text. *Be sure you remember your change-code -- the HYPERAUTHOR is merciless and won't ever let you access your lesson again if you forget it.* You can see that your change-code should also be kept secret, so that others cannot accidentally or maliciously change your lesson without permission. Again, it is difficult to change a change-code, so don't let it get out!

If the lesson is just being created, the interchange would look like this so far:

EDITOR READY *author types*
AUTHOR MODE *this*
WHAT LESSON? *test*
WHAT CHANGE CODE? *FEB19*
LESSON DOES NOT EXIST. NEW LESSON?

The last line indicates that the EDITOR did not find the lesson on the disk unit, and it requires a *YES or NO* answer about whether this is a new lesson or not. It is new, so type "yes". If you know that your lesson previously existed, first check the

spelling of the name, to be sure you typed it correctly. Then check the change-code. If one of them was wrong, say "no" and the HYPERAUTHOR will let you try again. If both were correctly spelled, there is a chance that the TUTOR text of your lesson has been *idled* automatically by the computer because you have not used it for several days (the policy now is five days). In such cases, you may use the GETFILE section of this document to determine how to retrieve your file. If you knew that your lesson was not going to be used for a few days and had the foresight to SAVE it on a magnetic tape, you can use the RESTORE lesson at this point to get it back. In either case, type *%DROP*, wait for the EDITOR to print its error message, and type *END* to get back to HYPERTUTOR, then take appropriate action.

If the author is accessing a previously created lesson, s/he follows the same procedure with lesson name and change-code, but the computer responds (for example):

```
 231 LINES READ
?
```

The EDITOR maintains a *workspace* of its own while an author is working on a lesson. It reads the permanent TUTOR text of a lesson into that workspace at the beginning of a NYPERAUTHOR run, lets the author make corrections and additions, and then writes the workspace back to disk storage before a condensed student copy is created for the lesson directory. When the text is completely read into the workspace, the count of its lines is printed and it's up to the author to determine what is to be done next.

EDITOR COMMANDS
Things to make life easier.

The entering of hundreds of lines of TUTOR commands into the lesson file can be an arduous task, even for an experienced programmer. So the EDITOR has a number of commands to make things easier. These are not part of the lesson, but are purely instructions telling the EDITOR how to manipulate the lesson text. In illustrations which follow, all *EDITOR commands* will be shaded (they are not a part of the TUTOR text, and this should remind you). All TUTOR commands will be un-shaded.

The EDITOR requires that you assign a permanent *line number* to each line of text entered into the computer. These line numbers need not be sequential, but will be sorted into ascending order by the EDITOR to determine the order of commands in the finished program. (The PLATO-IV editor, in Urbana, does not need permanent line numbers, and you will find this to be an exception -- most editors will require numbers.) To enter the text of a new lesson, one types line numbers, each followed by a period and then the actual text of that line. An example:

EDITOR commands and interaction shaded

```
EDITOR READY
AUTHOR MODE
WHAT LESSON? test
WHAT CHANGE CODE? jsfeb19
LESSON DOES NOT EXIST. NEW LESSON? yes
? 10.name       test
? 20.title      a test lesson
? 30.unit       one
? 40.at         503
? 50.write      hello, this is a lesson!
stop
```

TUTOR-IV commands unshaded

what the user types is italicised

The line numbers really tell EDITOR where in the workspace the next line of text is to be inserted. [If there is any question about this, try the example listed above on the computer as soon as you can. Be sure to observe correct spacing in the TUTOR commands -- as an alternative to the spaces between the *command* portion of a line and its *tag* , you may wish to use a semi-colon (;). Line 10, for example, could be typed:

```
? 10.name;test
```

Mistakes can be corrected by retyping the line at any time. [If you accidentally type the same line number twice with different text, the second one will, of cour be the one retained.] If you later find

that you need to insert more lines between two existing lines, pick a number which would lie between them and type that line -- the EDITOR will insert it in numeric order. This is why the numbers in the example on the last page are incremented by tens -- so there will be room between them for more additions and changes.

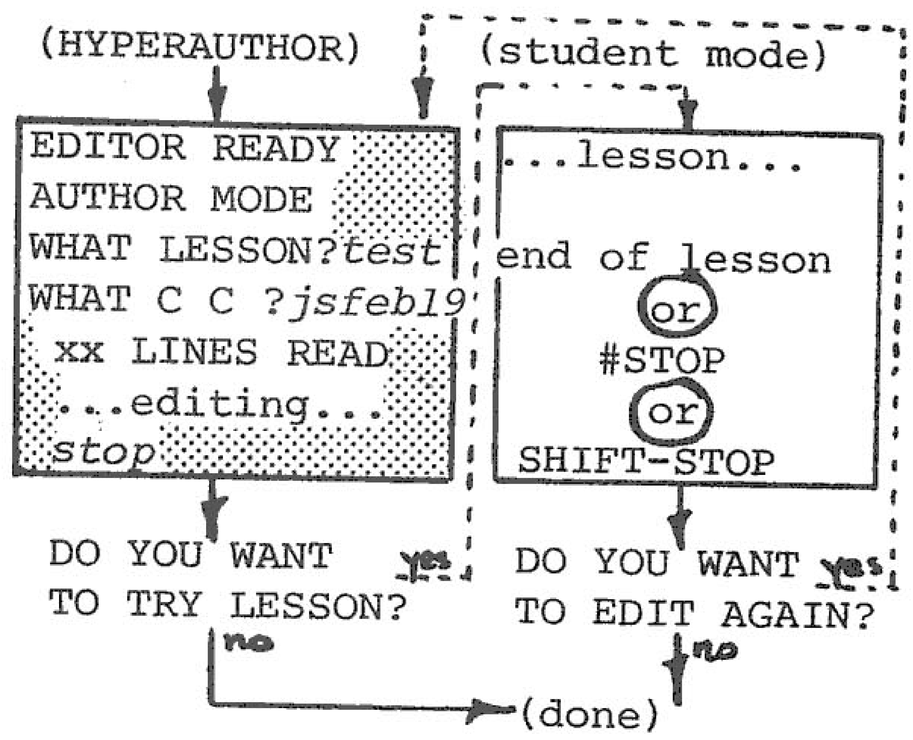
After a number of such changes and insertions have been made, it is wise to list part or all of the workspace to be sure things are going well. The command *LIST*, abbreviated *L*, will list any line or sequence of lines. It has three forms:

? *LIST*
? *LIST,40*
? *LIST,40-80*

The first of these will list the whole TUTOR text of the lesson. The second is used to list an individual line (in this case, line 40), and the third to list a sequence of lines (40 thru 80 in this sample). These commands will be of most use when authors are using cathode-ray-tube (CRT) terminals, on which only a few lines can be shown at a time.

A full paper listing can be made of any lesson at the time it is condensed, by simply telling HYPERAUTHOR to do so. These listings are printed at high speed, either at the Vogelback Computing Center, or at any of the high-speed remote terminals.

You now know all you need to know to enter a lesson, correct errors and add new lines to it. To leave the EDITOR and test the lesson, you need to know about *STOP*. When the EDITOR is told to STOP, it first copies the workspace to "permanent" disk storage, replacing the previous version. It is at this point that a copy is *condensed* and added to the student-mode lesson directory. The HYPERAUTHOR then asks you whether you wish to try the lesson in student-mode. If you do, you may be able to detect errors or unintended actions in the lesson. Once you've found a number of these errors, or have completed the lesson, you end student mode (#STOP or SHIFT-STOP on PLATO terminals) and return to HYPERAUTHOR to edit again. If you give the same lesson name and change code, EDITOR reads the lesson into its workspace and you cycle as many times as required to fix the lesson.



[For those worried about the cost of computer time; it saves a lot of time if you make as many additions and changes as possible before typing *stop* , then find as many errors as possible in student-mode before returning to edit. It is the reading and writing of the lesson into the workspace, and the condensation process which are most expensive and therefore should be done as seldom as possible.]

MORE COMPLICATED EDITING

An author often wishes to find some particular word or phrase in his TUTOR text and does not remember where it was used in the lesson. The SEARCH command finds it:

? SEARCH"cyanide"
? SEARCH"cyanide" 1000-2000

The first will report back by listing all lines in which the word "cyanide" appears in the entire lesson. The second will list any line between 1000 and 2000 which contains the word. *Any* group of characters (or *string*) can be found using SEARCH. Often, authors will use slashes (/) rather than quotes (").

A second type of SEARCH is used in certain situations, and is less expensive. It's called an *anchored search,* and is specified:

ASEARCH"unit"

It is called "anchored" because it will find occurrances of the string you specify *only when they begin in column one of a line.* It is extremely useful for searching for certain commands, such as "unit", above. In a lesson with several units, the search might give output:

```
10 UNIT      MARVEL
105 UNIT      TWIXT
507 UNIT      ZERO
607 UNIT      LEWIE
```

indicating the first line number in each unit. Other commands which just happen to contain the word "unit", like:

```
777 WRITE     THIS UNIT IS A...
```

wouldn't be reported by an ASEARCH.

Sometimes an author will find that a relatively small portion of a line or a group of lines is erroneous (the latter case might be where the author realized that he has misspelled a word throughout a lesson). Instead of retyping the entire line, or all occurrances of the word in the lesson, the author may use the REPLACE command.

REPLACE"x"y"

REPLACE"x"y" 40-80

Like SEARCH, REPLACE first finds the string you've specified -- it then will replace the specified string with a *new* string of characters. Like SEARCH, the REPLACE command can be limited to a single line, or to a range of lines. In the examples above, all "x"s would be replaced by "y"s. Because REPLACE deals with strings of characters, one must be careful in specifying replacements -- a common error is to forget that a set of characters like "the" is often found in other words. For instance:

REPLACE"the"these"

would cause the line below...

```
40 WRITE     Here is some theory...
```

...to become...

```
40 WRITE     Here is some theseory...
```

As each line in a lesson's TUTOR text requires a line number, which must be in ascending order, the author will sometimes find it desirable to reset all lines to a uniform sequence -- most often this is desired after a lot of insertions have been made. The RESEQ command (which stands for RESEQUENCE) will renumber all lines in a lesson, incrementing them by tens. It is very desirable to obtain a printed copy of the lesson after resequencing. The lesson:

```
10 name      test
11 title     a test lesson
18 unit      one
20 at        503
45 write     hello, this is my
46           test lesson...
```

...would become...

```
10 name      test
20 title     a test lesson
30 unit      one
40 at        503
50 write     hello, this is my
60           test lesson...
```

...upon resequencing.

Often an author will find that s/he needs a copy of certain lines of text in another place in the lesson text. COPY will copy a single line, or a set of lines, leaving the originals intact.

COPY 40,93

COPY 40-80,6000,5

The first form copies a single line to the specified new location -- line 40 will be copied and placed at line 93. If line 93 previously existed, the new one will replace it, of course. The second form copies lines 40 thru 80 to begin at line 6000, incrementing the new lines by 5. The increment is optional, and will be assumed 10 if not specified. One must be very careful with COPY because it will always replace existing lines at the new location with the copied lines.

At times an author will desire to com-

pletely MOVE a section of text to a new location in the lesson. This is done using MOVE, whose format is exactly the same as COPY -- it essentially performs a COPY and then deletes the original lines from the lesson.

MOVE 40,93

would bodily move line 40 to new position 93.

When long sections of text are to be entered continuously, the typing of line numbers may be a great convenience. They may be omitted under the following command, TEXT. While the EDITOR is in TEXT mode, everything typed goes directly into the TUTOR text of the lesson. No LISTing, COPYing, etc. may be done while EDITOR is in TEXT mode! One escapes the TEXT mode by typing *%E* :

```
EDITOR READY
AUTHOR MODE
WHAT LESSON? test
WHAT CHANGE CODE? jsfeb19
LESSON DOES NOT EXIST. NEW LESSON?yes
? text
? name;test
? title;a second test lesson
? unit;one
? at;503
? write;this is my second
? ;lesson, entered in TEXT!
? %E
? list
10 NAME      TEST
20 TITLE     A SECOND TEST LESSON
30 UNIT      ONE
40 AT        503
50 WRITE     THIS IS MY SECOND
60           LESSON, ENTERED IN TEXT!
? stop
```

note: no line nos.

this section in TEXT mode

It is very important that you remember to type %E to terminate text mode!

TEXT can be used in two ways: 1) to add a series of lines at the physical end of a lesson, or 2) to add lines in the middle of a lesson. When one types TEXT by itself, lines are added at the end -- when you type TEXT, followed by a line number, insertion begins at the line you specify. TEXT increments its line numbers by ten unless you include a second number after the starting line number:

TEXT 6000,5

for instance, would begin adding lines at 6000, and would increment numbers by 5.

DELETING

It is sometimes necessary to go back to delete whole sections of text from a lesson. The DELETE command has the same format as LIST:

DELETE 98

DELETE 100-199

The first form deletes a single line -- the second deletes a range of lines.

CATASTROPHES

Because the EDITOR reads a copy of the lesson text into its workspace at the beginning of a HYPERAUTHOR session, it is possible to recover from catastrophes (like accidentally deleting most of a lesson) by directing the EDITOR not to move the workspace back to permanent storage at the end of the session. Do this by typing:

END

as soon as the error is discovered. Control is immediately returned to the HYPERTUTOR.

MORE INFORMATION

The EDITOR is maintained by the Vogelback Computing Center, which publishes a guide for users. It has a HELP command which lists the commands available to users, and an author may request a shorthand explanation of any command by typing the word EXPLAIN, followed by the name of the command:

EXPLAIN DELETE

These explanations are short, and do not include information for novice users, and we recommend that new users first try one of the examples we've given.

ABOUT C.A.I.

Computer-Aided-Instruction (CAI) takes many forms: *games, simulations, inquiry mode, tutorials, programmed instruction or drill and practice.* These may be combined with other computer science techniques, *information retrieval, statistical manipulations, graphics* and other special-purpose programming systems. This chapter will outline some forms these techniques might take, and will give some examples. It is followed by a short annotated bibligraphy.

The TUTOR-IV language is much easier to use (and more powerful) than other CAI languages. Although it would be easy to program information retrieval or statistical operations in TUTOR, the computer is shared among *many* students in CAI systems, and there is also not enough flexible storage space on most CAI systems to allow such computer-intensive uses. Each student is provided with a small storage space, and many programs may simultaneously access common storage space for communications purposes -- this is plenty for most of the CAI applications discussed here. The HYPERTUTOR (as opposed to PLATO) provides direct links to programs in other languages for those who need them at appropriate moments.

GAMES

Computerized games can be used to teach kids, but most of them are sheer diversion. In *spacewar*, for instance, two opponents see simulated spaceships on their screens. By pressing appropriate keys, they maneuver their ships and try to "hit" each other with space rays. A number of variations exist; the most popular one on PLATO is *dogfight*, in which the spaceships have been traded for world-war I airplanes. Games are used to great advantage with young kids in the elementary math program on the PLATO system. In one game, kids get to choose how many bees to put into a beehive on the screen. They then press "i" to put bees into the hive and "o" to take them out -- they have to learn to subtract in order to tell how many bees are (invisible) inside the hive by counting how many are outside. And the kids learn to count, assign place-values, etc. from more games.

SIMULATIONS

Simulations can serve as learning tools by acting as complex games in which the learner must guess the rules of the game or determine how to use those rules to achieve a certain outcome. The *population* simulations of the Population Dynamics Group at the University of Illinois are used in this way. The students must manipulate variables which affect the population of a country and observe the outcomes as part of the learning situation. Simulations have also been used in teaching electrical engineers about the properties of electronic circuits -- they change the values of various system components and observe the outcome in terms of the actual performance of the circuit.

INQUIRY MODE

This is the "best" of the CAI modes because the student gets to ask the questions. The lesson usually sets a context and the student takes it from there. A few good inquiry mode lessons exist on PLATO -- one of those is an organic chemistry lab in which the student asks questions about an unknown substance and the computer gives the answers. The student must work toward an identification of the unknown, and the computer doesn't even prompt unless asked. Inquiry mode varies quite a bit, but in general it means that the student asks the questions -- sometimes the student is prompted by a teacher and sometimes there's a workbook giving useful hints -- what's important is the student control. (Note that a general context for the inquiry is always set by the program -- the student can't just ask any question!)

TUTORIAL

The ideal teaching situation is one in which the student has a private tutor to answer questions and give guidance on all matters. The word *tutorial* has come to mean (in CAI terms) an interaction in which the computer selects the next step for the student, corrects his errors and determines what remediation is necessary. A perfect tutorial would contain enough material to satisfy the slowest or the fastest student, and would be able to pick the appropriate time to present it. It is extremely difficult to write good tutorial lessons. Though most CAI experts define the word rather strictly, I like to believe that a true tutorial includes material written in both inquiry and programmed instruction modes, where appropriate.

DRILL & PRACTICE

Patrick Suppes, of Stanford University, has been the most visible proponent of the drill & practice mode. In this mode, the student is drilled on a specific skill, until s/he masters it. A subject area must be mapped completely before a drill & practice lesson can be written for it. D & P lessons may be pieced together in sophisticated ways -- students might be branched to "faster" or "slower" lessons, tested, or tutored as part of the program. No one claims that any one CAI strategy is the total answer to teaching via a computer.

PROGRAMMED INSTRUCTION

The term *programmed instruction* is abused. Strictly used, it applies to a mode of interaction in which a problem is presented to the student which requires a one-word answer. The student answers, and then is allowed to see the correct answer -- immediate knowledge of the correctness of his answer is an integral part of P.I. Most programmed instruction is carried out by programmed textbooks, on paper. The computer can be programmed to carry out such a strategy, but is much more expensive than paper, and should be used in more complicated strategies. The TUTOR language is well suited for all of the strategies outlined above.

FURTHER READINGS

TICCIT system:

"An Overview of the TICCIT Program", Mitre Corp., 1820 Dolly Madison Blvd., McLean, VA 22101, January, 1974

Bunderson, C. Victor, "The TICCIT Project: Design Strategy for Educational Innovation", Tech. Report 4, I.C.U.E., W-164 STAD, Brigham Young University, Provo, Utah 84602, Sep. 1973

PLATO-IV system:

Bitzer, Donald L. et al, "Computer-based Science Education" CERL Report X-37, May, 1973 [Computer-based Education Research Lab, Univ. of Illinois, Urbana IL 61801]

PLANIT:

Frye, Charles, "CAI Languages: Capabilities and Applications", *Datamation*, 14(9), Sept., 1968, pp. 34-37

General:

Hicks, Bruce & Hunka, S., *The Teacher and the Computer*, W. B. Saunders, Philadelphia, 1972

Atkinson, Richard C. & Wilson, H.A., *Computer-Assisted Instruction: A Book of Readings*, Academic Press, New York, 1969

Holtzman, Wayne H., *Computer-Assisted Instruction, Testing, and Guidance*, Harper & Row, New York, 1970

Anastasio, Ernest J. & Morgan, Judith, *Factors Inhibiting the use of Computers in Instruction*, EDUCOM, Princeton, 1972

HYPERCOMMANDS

On the PLATO-IV system a TUTOR program (or lesson) stands by itself, and is accessed by name. On HYPERTUTOR, a lesson is located not only by name but by additional information which is available in a browsing mode. A few prefatory commands (called HYPER-COMMANDS) appear at the beginning of a HYPERTUTOR lesson -- they are used by HYPERTUTOR to position your lesson, and by students to determine what a lesson is about. These commands include: NAME, TITLE, AUTHOR, SUBJECT, STATUS, LEVEL, PUBLIC, PRIVATE and PASSWORD, plus the "links" for the lesson. These commands are always the first to be entered when a new lesson is created, but may be changed if required at a later date. Example:

```
EDITOR READY
AUTHOR MODE
WHAT LESSON? test
WHAT CHANGE CODE? jsfeb19
LESSON DOES NOT EXIST. NEW LESSON? yes
? 10.name       test
? 20.title      a test lesson
? 30.author     jim schuyler (492-5367)
? 40.subject    a short introduction to
? 50.           fractions.
? 60.status     incomplete
? 70.level      high school
? 80.password   zorch
? 90.unit       one
?   .etc...
```

The NAME given in the NAME command is almost always the same as the one given to the HYPERAUTHOR (although it may differ if you wish). If desired, it may extend to ten characters, in contrast to the seven characters HYPERAUTHOR will accept. The TITLE must be a single line, and is displayed in browsing mode. It is best to limit the title to 50 characters or less. The SUBJECT command gives a broader description of the lesson, and may occupy up to eight lines. The other commands are explained in the COMMAND SUMMARY section.

Since the essence of the HYPERTEXT configuration is the links among the lessons, special HYPERCOMMANDS have been created to describe these relations. Most authors will use UP, DOWN, RIGHT and LEFT, with the following meanings:

UP = more difficult
DOWN = easier
RIGHT = a related subject area
LEFT = a related subject area

Each of these linking HYPERCOMMANDS will specify a single lesson name with the indicated characteristics. For instance, we might add links to the example in the left column:

```
? 81.up       numtheory
? 82.down     division
? 83.right    reduction
```

In the browsing mode, when the student types UP, s/he will be branched to the lesson named "numtheory", where browsing will continue.

It is our plan (for the near future) to allow links to be created by students, or given non-standard names by authors. For instance, an author would be able to create HARDER or EASIER links instead of UP and DOWN.

If PRIVATE and PUBLIC are not used in a lesson, the lesson will be declared PRIVATE by HYPERTUTOR (automatically). At any future time, the author may insert a PUBLIC command to put the lesson name on the browsing list. Again, at some future time, there will be a mandatory review procedure for lessons which are to be PUBLIC -- during some period of time they will be placed on a "probationary" PUBLIC list, and may be tried by students or other authors. When they have received enough positive "votes", they will be added to a more permanent PUBLIC listing.

LEARNING ABOUT TUTOR

The primary reason authors have little trouble mastering the TUTOR-IV programming language is that the commands one gives to the computer are quite similar to the lexicon one would use in English to accomplish the same purpose. Another reason may be that, in contrast to other programming languages, the progression of TUTOR statements is quite logical; the sequence of the commands follows rather closely the conceptual order of events which the author has imagined for his lesson.

TUTOR commands to be processed together are grouped into *units*. These units all start with a UNIT command. All of the commands after a particular UNIT command, up to the next UNIT, are a part of that logical entity. Example:

```
unit      initial
at        503
write     welcome to a lesson on the
          names of the Watergate
          conspirators.
at        903
write     For our "records", type
          your name below:
arrow     1103
storea    nl,jcount
ok
next      begin
unit      begin
...etc...
```

In this sample there is a single unit (named "initial") and the first command in a second unit ("begin"). A reasonably satisfactory definition of a UNIT is "a set of commands intended to be processed together."

COMMAND AND TAG

Each line of a TUTOR program is composed of two parts, *command* and *tag*. The command always begins in column 1, immediately after the period which follows the editor line number.

```
10.name;test
```

The tag always begins in column 11, or immediately after the semicolon (if you use it to separate command and tag). The semicolon is interpreted by the EDITOR and is not officially a part of the TUTOR command! When the EDITOR later lists the line, you will find that it has been replaced by the appropriate number of spaces to make the tag appear in column 11. The tag position may vary from TUTOR system to TUTOR system.

Like most TUTOR commands, the UNIT command always has a tag. The UNITs in the example at left are "initial" and "begin" and are named in the tags of their respective UNIT commands. The tag on UNIT may consist of any combination of numbers and letters the author selects, as long as it does not exceed **eight (8)** characters in length, and as long as there is not already a UNIT by the name used. The reason for the last condition is understandable: if the computer is told to process next in a unit called "begin", there had better be only one unit by that name!

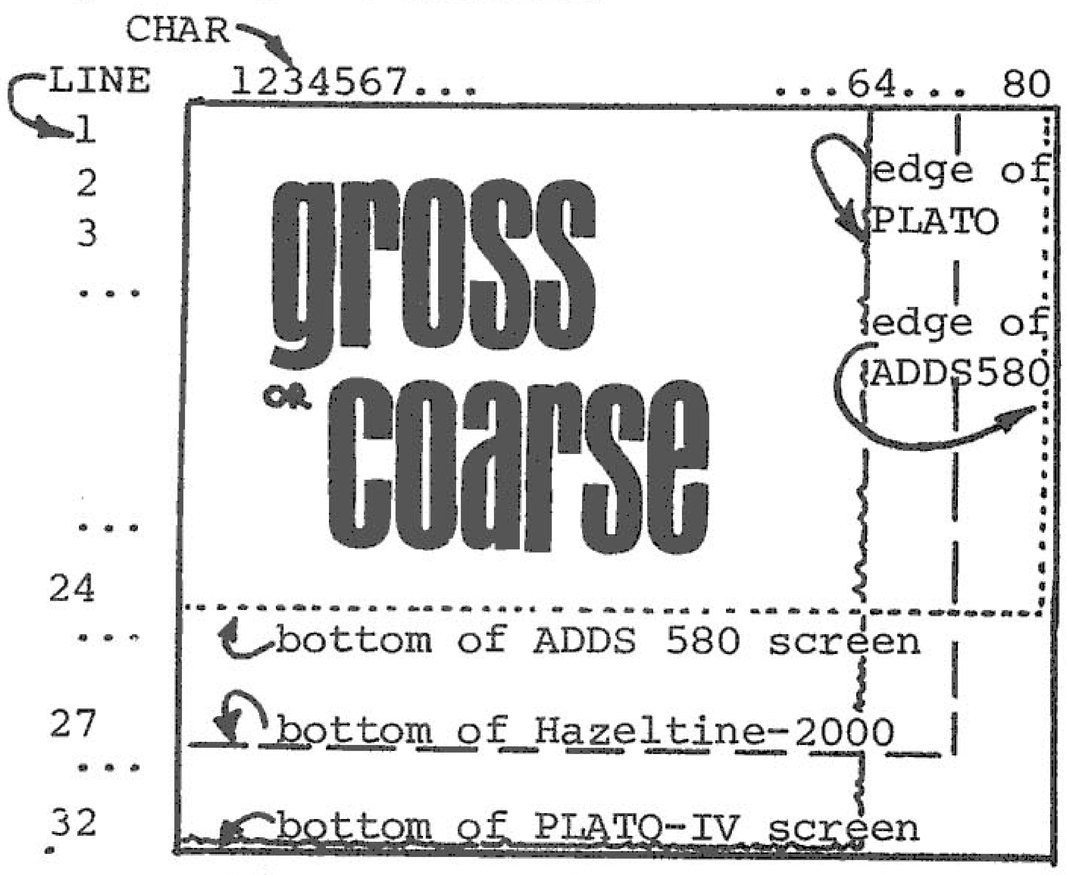
The NEXT command is used to specify the order in which units are to be processed. The NEXT command in the example specifies that "begin" is to be processed next. It can be used in many ways, to adjust the student's progress, or to branch depending upon the last error made.

UNIT and NEXT are the most basic sequencing commands. UNIT is really just a marker which gives a name to a set of commands; NEXT is actually processed when the student is running a lesson, and gives the computer instructions about which unit is to be processed next.

Two other important commands from the example are AT and WRITE, both controlling actions on the student's screen. They are almost always found together. AT specifies where information is to be written on the screen, and WRITE tells the computer what to write.

THE "AT" COMMAND

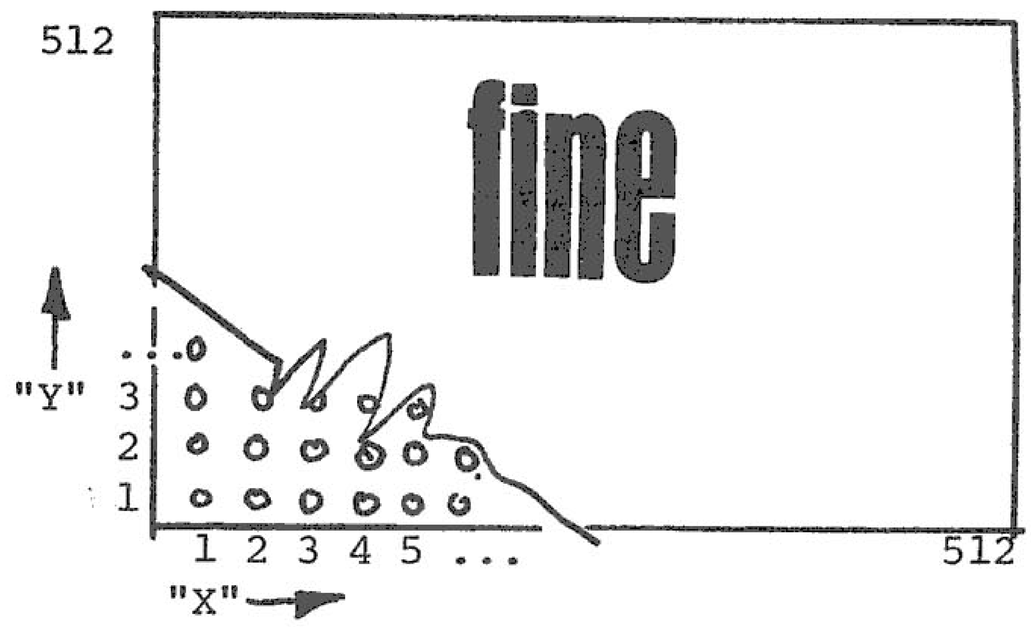
AT has as its tag a "screen address" in one of two possible formats. The first of these is called "gross" coordinates, and the latter, "fine" coordinates. In the gross-grid, a position is specified by LINE and CHARACTER.



One combines LINE and CHARACTER into a single number, the "hundreds" portion of which indicates the LINE, and the "units" and "tens" indicating the character position on that line. The number "903", for example, indicates line 9, character position 3. "1245", in a similar fashion, indicates line 12, and character position 45.

If an author is writing a lesson which is to run on any type of terminal, screen positions must be restricted to the lowest common screen size, 24 lines by 64 characters. Lessons which require the drawing of lines, or use of special characters, *must* run on PLATO-IV terminals, and may use the entire screen of that terminal.

On the PLATO-IV terminal, the "fine" coordinate system may be used to attain more control over the placement of text and lines on the screen. When one uses fine-grid coordinates, the first number specifies the "X" (horizontal) position on the screen, and the second specifies the "Y" (or vertical) position. Each of these numbers may range from 1 to 512. A change of a single unit in either the X or Y direction produces a change of ONE DOT in the appropriate direction on the student's screen.



A sample unit is shown below:

```
unit      first
at        1205
write     This lesson will introduce
          the metric system.
at        1505
write     Do you know anything about
          this system already?
arrow     1705
answer    yes
next      review
answer    no
next      second
```

The first AT specifies a screen position in gross-grid coordinates, at line 12, character 5. The WRITE command which then follows will begin writing text at that position. The second AT selects a lower position on line 15, where the second WRITE will place its text. In addition, the ARROW command (to be singled out in later sections) also specifies a screen position on line 17, for the "arrow" which indicates that the computer is waiting for the student's response. The figure on the next page shows what the student would see when running the unit above. Note that each of the WRITE commands was written as a paragraph, and that the HYPERTUTOR has

neatly paragraphed them on the screen, with a straight left margin.

```
06/01/74  15.55.00.


This lesson will introduce
the metric system.

Do you know anything about
this system already?
>
```

WRITE

Though the author of this example has used separate WRITE commands, one can always put several sentences, or even paragraphs into a single WRITE:

```
unit      first
at        1205
write     This lesson will introduce
          the metric system.

          Do you know anything about
          this system already?
arrow     1705
answer    yes
...etc...
```

The purpose of the WRITE command is to provide *text to be written on the screen*, and text is always displayed exactly as written by the author (except in cases where it would run off the edges of the screen, when unpredictable thing happen). On the PLATO-IV terminal, upper and lower case characters may be used: on other devices, the standard is usually upper case. Authors writing on CRT terminals may write lessons which will produce capital letters when run on PLATO terminals by inserting an up-arrow (↑) before any character which is to appear capitalized on PLATO terminals. The up-arrow is automatically removed when the lesson is run on normal CRT terminals.

Authors will note that printed versions of lessons which contain capital letters contain these up-arrows, because our printers do not have capital and lower case letters.

Combinations of AT and WRITE may be used to place text anywhere on the screen, in any order -- however, if an author wishes to produce lessons which will make any sense at all on Teletype (or other printing) terminals, it is best to at least WRITE in a logical order, so that the information would be readible if written down the left-hand margin of the page. [Because of the plethora of terminal types, we seldom suggest that a lesson be written in such a fashion -- more often it is possible to write *sections* of a lesson to run on the different devices, and then ask students to switch to CRT or printing terminal at appropriate times.]

ASKING QUESTIONS

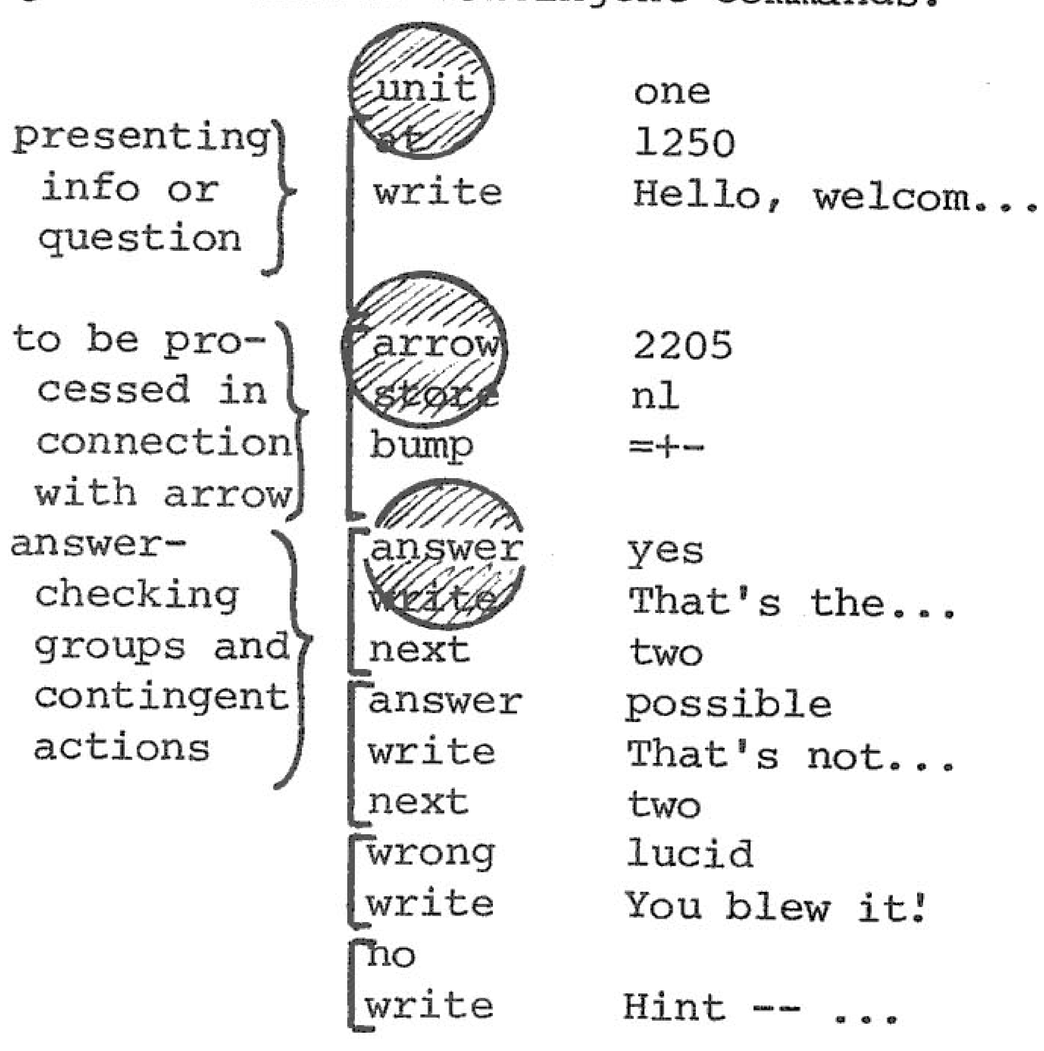
Just presenting information to the student on a computer terminal isn't enough to make a decent "lesson" -- one must be sure the student understands the subject, and this is done by asking questions. In the Inquiry mode, for instance, the student is the one asking the questions, but in all modes it is important that there be a two-way interaction. TUTOR and HYPERTUTOR signal the student that an answer (or other student action) is required by printing an arrow (>) on the terminal and waiting for student action.

The author of a lesson determines when a student response or action is required and places an ARROW command in the lesson at the appropriate spot. As indicated in the last section (on the AT command), an ARROW also has a screen position specified with it. When the student types an answer, the characters always appear to the right of that arrow on the screen. Every response to an ARROW must be processed and "judged" right or wrong (but

the judgement may be withheld from the student, if the author so specifies). When the student gives a CORRECT answer, the characters "ok" are written to the right of it. Incorrect answers (or an answer the author didn't anticipate) are marked "no". If desired, the author of a lesson may change these judgement words to two of her choice.

The ARROW command is intimately involved with a set of commands to be called answer-checking or judging commands, used to check the "answer" a student gives to a question. These include ANSWER (specifying a correct answer or set of synonymous answers, to be marked "ok"), WRONG (specifying an incorrect answer, judged "no" if it appears), OK (specifying that unanticipated answers are to be treated as correct answers) and NO (specifying that unanticipated answers are incorrect), and CONCEPT.

As a guide to understanding the interaction among various types of TUTOR commands during the judging of answers, we will now divide the "unit" concept, so that a unit consists of several sections: Unit-contingent, arrow-contingent and answer-contingent commands.



(Some units contain no ARROWs, but most will follow this organizational scheme.)

In the "presenting" section, there will be AT and WRITE commands, often with other appropriate "regular" commands which either perform calculations or show the calculated values of TUTOR variables. These put information on the screen, or present the question to be considered by the student. The ARROW will at times be followed by commands which store the student response (STORE appears in the example) or remove characters from it (BUMP, in the example). Then, a large number of answer-checking or judging commands may follow, in any order. When the student types an answer and presses the NEXT (or carriage-return) key, HYPERTUTOR will attempt to make a close match with the tags of the answer-checking commands. If a good match is found, HYPERTUTOR will process the "regular" commands (AT, WRITE, NEXT, JUMP, etc.) which follow the answer-checking command. These statements are said to be answer-contingent, because they are performed *only when their corresponding answer matches the student's answer*. This makes it possible to give hints or display remarks, and to branch to a new unit, depending upon the answer the student gives. If none of the ANSWER or WRONG commands yield a close match, any OK or NO command will be processed, and answer-contingent commands which follow may again be used to give hints or to branch to another unit.

If the judgement "no" is given, the student is forced to repeat the ARROW (and therefore, to answer again) until a correct answer is finally given. (It is always possible to count the number of incorrect answers a student gives and branch to remedial units at the appropriate time, but this isn't done in an "automatic" way.) When an "ok" answer is given, the student is free to go to the next unit -- if a NEXT command follows the answer-check which matched, s/he will go to the unit named in that command. If there is no NEXT command, the student will proceed to the next unit (in physical order) in the lesson. NEXT is only effective when the answer is judged "ok" and is fully treated in another section.

At this point in your TUTOR education, it would be appropriate to go to the computer and attempt your own program. That program might consist of a single unit (or short series of units) asking simple questions, like the sample below:

```
name      test
title     joe's test lesson
unit      shampoo
at        503
write     What is the capital of
          Illinois?
arrow     617
answer    Springfield
next      mass
wrong     Chicago
at        703
write     That's the largest city,
          but it isn't the capital.
wrong     (Rockford,Peoria)
write     Sounds like a guess to me.
no
at        703
write     Hint -- Spring follows
          Winter...
```

If a student were to try this lesson, the following might happen:

```
What is the capital of
Illinois?      >
```

Initial display
Student answers and pushes NEXT

```
What is the capital of
Illinois?      > Chicago   no
That's the largest city,
but it isn't the capital.
```

That's wrong, so student must answer again:

```
What is the capital of
Illinois?      > Springfield   ok

press NEXT
```

Now the student has answered correctly, and upon pressing NEXT, will be branched to the appropriate unit, "mass", as specified in the NEXT command which is contengent upon the answer "Springfield".

On the HYPERTUTOR system, the message "PRESS NEXT" (or the appropriate key name for the terminal being used) will appear at the bottom of the screen, to prompt the student.

Because the student cannot proceed until an "ok" answer has been made, you (the author) must make your questions very clear, and should always provide a "universal" OK or NO command to give a clear hint.

WITHHOLDING THE "OK" OR "NO"

Often an author wishes to either eliminate or change the "ok" and "no" judgements. To totally eliminate them, insert:

```
specs     nookno
```

between the ARROW and the first answer-checking command. A WRONG answer will still cause repetition of the ARROW, but the "no" will be supressed. And when a correct answer is given, the "ok" is also suppressed.

One changes the judgement words with:

```
okword    right
noword    wrong
```

using any word (or characters) up to seven in length, as the tag.

LEVELS OF RESPONSE-CHECKING

HYPERTUTOR chacks author-specified responses against the student's response in a number of ways:

1. exact matches,
2. spelling errors marked "ok",
3. affixes marked "ok",
4. keyword matches (searching for an occurrance of the author-specified answer *within* the student's answer, like "yes" within "yesterday".)
5. synonym vocabularies, and
6. other extra HYPERTUTOR methods.

A match at *any level* will produce the appropriate "ok" or "no" judgement (depending on the answer-checking command which was used).

Authors may select the type(s) of matching operation to be attempted on student answers by mixing the various types of answer-checking commands:

1. EXACT,
2. ANSWER and WRONG,
3. OK and NO,
4. ANSV, WRONGV, ANSVA and WRONGVA.

EXACT is an answer-checking command which is matched only when the student's answer is exactly the same as the one specified by the author of the lesson. (Exceptions -- the student may insert extra blanks or punctuation when HYPERTUTOR is checking, but not on PLATO.) ANSWER and WRONG will check spelling, and will allow misspelled words, but on PLATO, misspellings are judged "no" and the student must try again. Affixes, in certain cases, will also be judged "ok". To allow "keyword" searches, an author specifies:

```
specs     okextra
```

on PLATO -- HYPERTUTOR will always perform keyword searches when ANSWER and WRONG are used. And finally, on PLATO, OK or NO commands are used to signal that unanticipated answers are to be judged "ok" or "no" -- on HYPERTUTOR, OK or NO may contain answer-checking information in their tags, and will be checked against special lexical information in a system-resident lexicon.

As an example, a unit is presented below which contains a mixture of answer-checking commands:

```
unit      choice
write     Choose the best answer from
          the following list.  The
          iguana is a:
           A. Fish
           B. Mammal
           C. Reptile
           D. Amphibian
arrow     1001
specs     bumpshift,okextra
exact     c
write     Absolutely correct!
answer    reptile
write     Absolutely correct!
wrong     (fish,mammal,amphibian,a,b,d)
write     The iguana is a desert lizard.
no
write     Please answer a, b, c or d.
```

An answer of exactly "c" or an answer containing the word "reptile" (capitalized or not) will be judged "ok". Other answers from the list will be judged "no", and any answer not in the list will be given a special message.

BRANCHING

Several commands may be used to specify the sequence of units in a lesson. Sequencing may be varied to suit individual student differences, or it may be the same for all students who take a particular lesson. The branching commands differ in whether they erase the screen or not, and in when they are actually performed. The commands to be discussed in this section are:

NEXT
NEXTNOW
JUMP
JOIN
GOTO

NEXT is used to tell HYPERTUTOR the name of the unit to be processed when the current unit finishes. If there is an ARROW in the current unit, the NEXT unit is not processed until the student's answer is judged "ok".

Thus, a NEXT command may be placed in several positions:

1. It may appear in the initial section of a unit, before any ARROW, in which case the NEXT unit is set but not accessed until the question has been answered...

```
unit      xxx
next      super
...
arrow     1250
...
```

2. It may appear after an answer-checking command which specifies a correct answer...

```
unit      xxx
...
arrow     1250
answer    correct
next      super
...
```

...in which case it is executed when the correct answer is given (same as case 1.)

3. It may appear after an answer-checking command which specifies an incorrect answer...

```
unit      xxx
...
arrow     1250
answer    correct
...
wrong     zilch
next      super
...
```

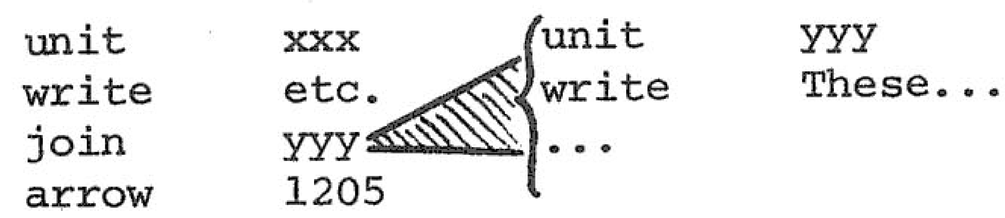
...in which case, no branching is done until a correct answer is given, but then the next unit is "super".

You can see that even though NEXT may appear in a number of contexts, it always has the same effect, that of specifying the NEXT unit to be executed when the current unit is done.

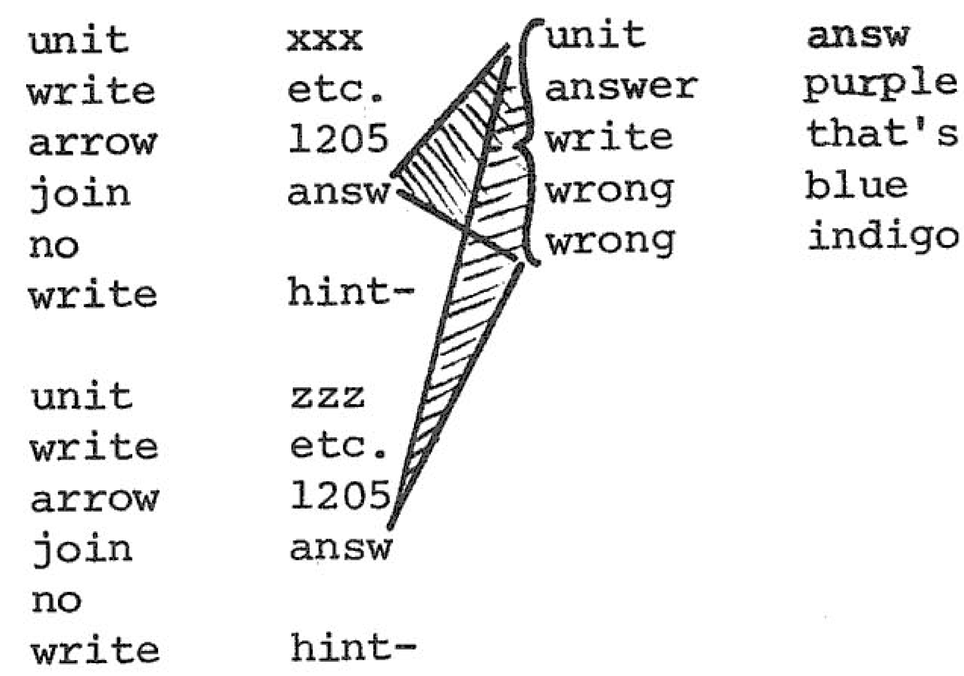
NEXTNOW is used to force the processing of a unit before the student answers correctly. It almost always appears after a WRONG or NO command. The student must always press NEXT (or appropriate carriage-return key) to go on.

JUMP will cause an immediate transfer to a new unit. The screen is erased immediately (the student doesn't need to press NEXT). NEXT, NEXTNOW and JUMP all reset something called the "base" unit pointer, indicating the current base of operations for the student. The commands which follow do not reset the base unit.

JOIN is used when an author needs to use commands from another unit, then return to continue processing in the base unit. Another way of thinking about JOIN is to imagine that all commands in the JOINed unit are physically "inserted" in place of the JOIN command itself:



JOIN is often used to connect to a unit which contains answer-checking commands needed in a number of units:



Those who already know something about programming will recognize this as a "subroutine" structure. Any unit in a lesson may be JOINed, whether it is used as a regular base or not. JOIN does not erase the screen. JOINed units may in turn JOIN other units, and in fact a JOINed unit may JOIN itself. This may continue up to six levels deep.

GOTO is a branching command used in situations where the screen is not to be erased and no return to the original unit is desired by the author. GOTO, however, does not change the "base" unit, and HYPERTUTOR will continue to the indicated NEXT base unit when

the GOTOed units finish. Note that a JUMP in the GOTOed unit would terminate the process, because JUMP resets the base unit immediately. The following table summarizes the names and actions of TUTOR branching commands:

TUTOR
branching

name	action	waits?	new base?
NEXT	sets next base unit name erases when NEXT pressed	for "ok" & NEXT	yes
NEXTNOW	sets next base unit name erases when NEXT pressed	for NEXT key	yes
JUMP	erases and jumps immediately	no	yes
JOIN	"inserts" commands from unit then returns to next command	no	no
GOTO	no erase, jumps immediately to new unit. base unit unaffected	no	no

Branching commands may be used to link units together in many ways. The most common way is to use NEXT, in combination with answer-checking commands, to steer a course thru the lesson. When the author desires immediate branches upon student answers, s/he usually uses JUMP instead of NEXT in answer-contingent situations:

```
unit      xxx
write     etc...
...
arrow     1205
answer    purple
jump      super
wrong     blue
...
```

(The JUMP, causes an immediate clearing of the screen and execution in unit "super", rather than the wait which would have been caused by NEXT or NEXTNOW.

CALCULATING

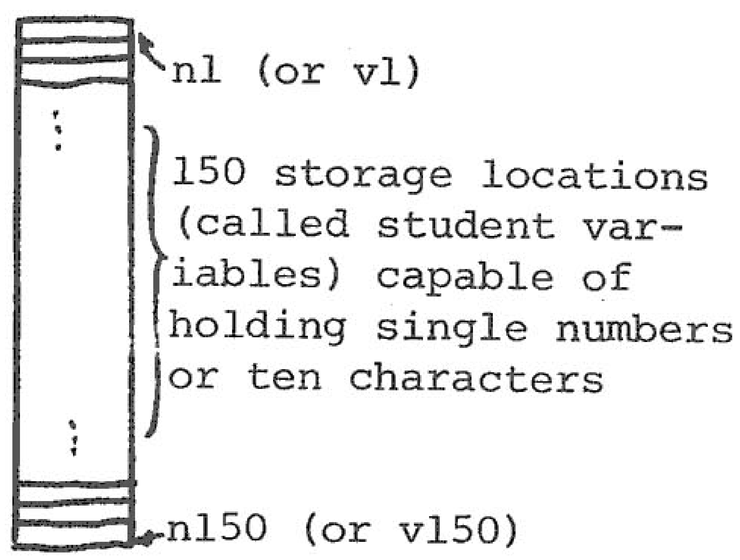
HYPERTUTOR (and TUTOR) provide a set of 150 storage locations for each student using the computer. These can be used to store intermediate results from calculations, and are preserved from session to session, though they are not commonly used for permanent data storage because they are reused by each lesson the student runs. [Permanent storage is called COMMON, and is separate from these 150 *student variables*.]

STORAGE FORMATS

Each of the 150 storage locations (called a *variable* or *student variable*), can hold a single number, or it can hold ten characters (in the form of a number, in fact). The lesson's author differentiates between uses of a variable by appending a format indicator ("v" or "n") to the variable number. "V" indicates that a variable will be used to store a "rational" number, requiring fractional parts. "N" indicates storage of an integer (no fractional part). For example, the number 3.5 must be stored in a "V" type variable, and the number 3 can be stored in either, because it can also be represented as 3.0 . Attempting to store a rational such as 3.5 in an "N" type variable will result in the rounding of the value and subsequent storage as an integer. 3.5 would be stored as 4.0, for example. [Note: Fortran, and other commonly used languages, would simply truncate or discard the fractional part, leaving the result 3.0 rather than 4.0, and programmers should take note!]

VARIABLE NUMBERS

References to variables contain two parts, the format (described above, which is either "V" or "N") and the variable number. A reference to variable "1", intended for use as an integer, would be coded "n1". A reference to a rational, stored in the same location, would be coded "v1". The range of possible names is therefore "n1" thru "n150" and "v1" thru "v150". Because "n1" and "v1" are physically the same location in storage, though referenced in different formats, it is usually unwise to use both names within the same lesson, though nothing prohibits this.



ASSIGNMENT OF VALUES

If a value is to be stored-away in a student variable it is always done via *assignment* in a CALC-type command or an expression. For example:

```
calc      n1 = 7
```

would store the integer value "7" in variable 1. The PLATO system uses the assignment-arrow (←) rather than the equal sign, and this change has to be made in lessons to be transferred to PLATO. Programs transferred from PLATO to HYPERTUTOR will have an exclamation point in place of the assignment arrow, but will execute properly. The right side of a CALC (the part after the "=" or arrow) may actually be any valid TUTOR expression. Expressions are built up from numbers (such as "7", "98.6", etc.) and from student variable names, combined with the normal arithmetic operators (+, -, *, and /). There are strict rules for forming expressions:

1. A number or student variable by itself is an expression:
   n1
   v77
   44.6
   ...and can therefore be assigned to a storage location:
   calc n99=9056
2. Two numbers or variables may be combined with an operator:
   n1+n44
   n55/6.3
   44 * n40
3. Because the entities described by "1" and "2" above are also expressions, we can inductively define any combination of these expressions and an operator as an expression:
   n1+n44+n37+5
   n1+n44+6 * v16
4. Because operators can be combined as in the last example above, definite precedence rules must be established to cover the order of evaluation of operators:
   a) exponentiation is always done first,
   b) any multiplications in the expression, then
   c) divisions, and finally
   d) additions and subtractions, in left to right order.

   Examples:
   n1+v66 * 6.3 --- v66 * 6.3 is done first, because of rule 4.b, then the addition is done.
   n1+n6-n7+n44 --- done in left to right order because of rule 4.d
   n1+6 / n2+6 --- surprisingly, 6/n2 is evaluated first because of rule 4.c, then the additions are performed, as if written: $n1+(6/n2)+6$
   n1 * 6 / n2 * 6 --- multiplication is done first, on n1 * 6, then on n2 * 6, and finally the division, as if written:
   $$\frac{n1 * 6}{n2 * 6}$$
5. Anything put inside of parentheses will be evaluated first, beginning with the

"innermost" parentheses:

```
n1 * ( n44 - 6.3 * ( 44 + n7))
```

--- the 44+n7 term is evaluated first because it is innermost, and the result is then multiplied by 6.3 (because of rule 4.b) then subtracted from n44's value and finally multiplied by the value of n1.

There are some additional considerations for those who use subscripts or superscripts, and those who use functions, primarily dealing with the precedence of these operators. This is all covered in the description of the CALC command in the command summary.

Note: On the PLATO-IV terminal a special multiplication sign (x) exists and is commonly used. Because many other programming languages use an asterisk to indicate multiplication, HYPERTUTOR will accept asterisks as well. If an "x" is inserted, HYPERTUTOR will also accept this usage.

To reinforce the notion of *assignment*, let me recapitulate. When a CALC is encountered, the expression to the right of the assignment symbol is evaluated and the result is then stored in the student variable whose name appears to the left of that symbol. A favorite example which reinforces the notion of assignment is:

```
calc      n1 = n1 + 1
```

...which would cause the evaluation of n1+1 first ("old" value of n1 plus 1) and its subsequent storing back in the first student variable.

A host of mathematical functions is available to all authors, including *cosine, sine, log (base 10), natural log, absolute value, exponential, the greatest integer function and square root.* These functions are included as part of the expression, in a natural way:

```
calc      v77 = cos (3.1415) + 1.0
calc      v70 = 98.6 * ln (5.0)
calc      n7 = sin(3.14)/cos(3.14)
```

New functions needed by authors may be defined within individual lessons, and because each student variable can, in addition, be given author-defined names, these expressions are quite readable.

COUNTING APPLICATIONS

For simple counting applications, such as remembering how many errors a student has made, additional CALC-type commands exist in TUTOR. The ADD1 and SUB1 commands add and subtract "1", respectively, from a student variable. In the example below, assume that "n1" has initially been set to zero (initialized to zero, is what a programmer would say), and it is being used to count wrong answers:

```
unit      xxx
at        503
write     Which color is the complement of orange?
arrow     703
answer    green
no
add1      n1
```

Each time the student makes an incorrect answer (unanticipated), "n1" is incremented. Its value can later be permanently stored, or printed for the student to see.

Calculations of this sort can also be combined with answer-checking to give a variable a unique value depending upon the student's answer:

```
unit      xxx
write     Which is the complement?
arrow     703
answer    green
calc      n1 = 1
wrong     blue
calc      n1 = 2
wrong     yellow
calc      n1 = 3
no
calc      n1 = 4
```

Because CALC is a "regular" command, it may be used in answer-contingent situations such as those in the last example.

The reader should bear in mind that often there will be several ways of programming a desired action -- the last example is particularly susceptible to alternate approaches.

DEFINING NEW VARIABLE NAMES

Often an author has trouble remembering what item is stored in each of the student variables. The best advice is to prepare a "variable map" and insert it as a long comment at the beginning of the lesson:

```
unit        colors
*           variable map :
*           n1 = wrong answer count
*           n2 = # of questions asked
*           n3/4 = student's name
*           n5 = errors in part 1.
*           n6 = errors in part 2.
...etc...
```

(Comment lines begin with an asterisk in column 1, in place of a command, and are ignored by the computer.)

In addition, HYPERTUTOR lets the author define his own names for student variables (or for especially useful expressions). The DEFINE command does this, listing a new "name" and an expression for which it is to stand thruout the program.

```
define    x=v1,y=v2,z=v3
```

...would define three new names, "x", "y" and "z". Whenever "x" is referenced in an expression in the lesson, "v1" is actually used. For "y", v2 is used, and for "z", v3. A wrong-answer counter might be defined...

```
define    wrong=n4
```

...then used...

```
calc      wrong=wrong+1
```

...or...

```
addl      wrong
```

Because any expression may be used, some rather complex definitions may result:

```
define    x=cos(v1),
          y=x/7.2,
          z=(x+y)/v70
```

In the HYPERTUTOR system, DEFINEs are actually processed as the lesson is scanned for insertion into the lesson directory (at READIN or CONDENSING time) and an actual physical substitution of the appropriate characters is made in the expression. Thus:

```
calc      v6=z/7.2
```

...actually becomes...

```
calc      v6=((x+y)/v70)/7.2
```

[Note that parentheses are inserted on both sides of the defined expression, so that it is evaluated before any other parts of the expression.]

SHOWING VARIABLE VALUES

The value stored in a student variable may be shown on the screen. The SHOW command, which includes some simple formatting information, is used...

```
show      n4,6
```

expression or variable — number of spaces to use

```
show      v4,6.3
```

expression or variable — number of places to left/right of decimal

IMPORTANT NOTE: PLATO has recently changed specifications on SHOW, and HYPERTUTOR will be updated to correspond during the summer.

One can show defined expressions, or other calculated numbers, and numerics can be combined with writing, since SHOW continues at the current screen position:

```
at        503
write     You made
show      n1,4
write      mistakes in the quiz.
```

Because SHOW and WRITE do not reset the screen position, the sample would appear on a single line on the screen.

NUMERIC STUDENT RESPONSES

Storing and manipulating numeric responses is done differently from the checking of alphabetic responses. To store a numeric, a special command, STORE, is inserted between the ARROW command and the first answer-checking command. (Some answer-checking command must always be present, even if it is only the universal-"OK".) The STORE is performed after the student has typed an answer, and it assumed that the student's answer is a number, with no letters or other words included.

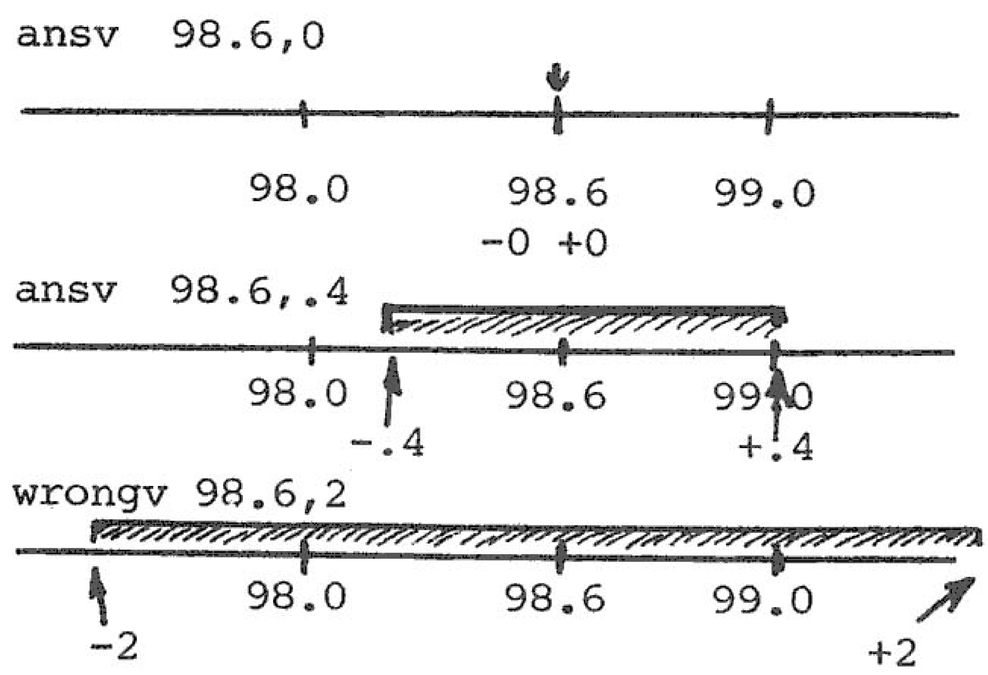
```
unit      xxx
write     Type a number --
arrow     503
store     v1
ok
at        603
write     Your number was
show      v1,4.3
next      yyy
```

The sample shown above will request a number, then store it in "v1" and show the result to the student, with four digits before the decimal and three after.

Just as often, an author wishes to determine whether the student has given a correct numeric answer (the example above simply accepts any number, and doesn't judge its correctness). This can be done with two numeric judging commands, ANSV and WRONGV. They differ only in the judgement made ("ok" for ANSV and "no" for WRONGV). ANSV and WRONGV are included along with other answer-judging commands, following the ARROW in a unit. In the tag, one specifies the correct answer (in terms of an expression or student variable), and either a range designator or a percentage designator. Anything within the indicated range of the correct answer, or anything within the indicated percentage of the correct answer will be judged accordingly.

```
unit      xxx
at        503
write     What is the normal temperature of the human body, taken
          under the tongue?
arrow     803
ansv      98.6,0
write     Absolutely correct!
ansv      98.6,.4
write     Close enough. 98.6 is the
          accepted figure.
wrongv    98.6,2
write     You are close. Try again.
no
write     Hint -- it's around 98°.
```

In this example, a sequence of ansv commands has been used to move progressively farther and farther from the correct answer. The first accepts only an exact answer of "98.6", the second accepts anything within ".4" of "98.6", and finally a wrongv will mark anything wrong which is within "2" of "98.6", but give a special hint. The chart below will help explain how this strategy works:



Because ANSV and WRONGV commands are processed as encountered, anything not trapped by the first one (namely, all exact answers) passes on to the second for evaluation, then if it is not within .4 of the correct answer, it passes to the third, etc.

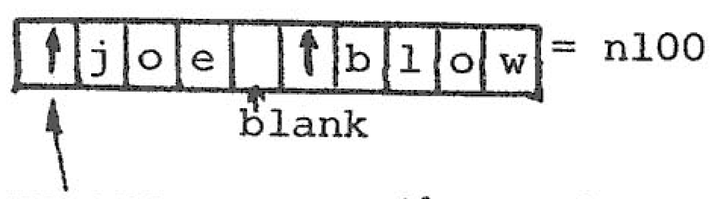
STORING CHARACTER RESPONSES

Earlier it was mentioned that a student variable can hold either a single number or ten characters (represented as a number). It is possible to store and show student

responses much as numeric responses are stored. The STOREA command (the "A" means alphabetic) is used:

```
unit      xxx
at        503
write     Type your name:
arrow     519
storea    n100,20
ok
at        603
write     Thank you,
showa     n100,20
write     .
```

The tag of STOREA specifies a student variable in which storing is to begin, then a character count. In the example, twenty characters will be stored. (If the student presses NEXT before typing that many characters, an end-of-line is stored, and a subsequent SHOWA will only show the number of characters the student typed.) Say the student is named "Joe Blow". HYPERTUTOR would store characters into n100 only, since it can hold up to ten:



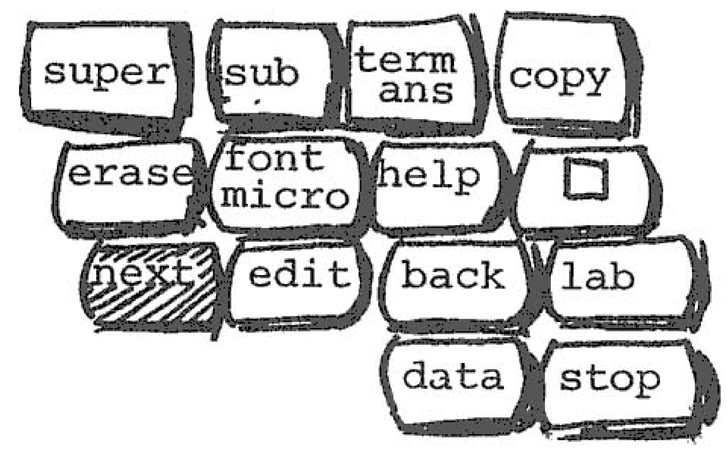
up-arrow means the next character is shifted ("J", "B")

If more than ten characters are to be stored, HYPERTUTOR will continue into subsequent locations (in the example, since a maximum of twenty will be stored, n100 and n101 will be the only locations used).

To display information stored in this fashion, one uses SHOWA ("A" for alphabetic). The first part of the SHOWA tag indicates the student variable to be shown, and the second part indicates the maximum number of characters to show. If fewer are present, only the actual number found are displayed.

FUNCTION KEYS

A special pad of extra keys is present on all PLATO-IV terminals, adding extra features for students and authors. The location of these keys is illustrated below:



The key marked "next" is the one which is pressed at the end of each answer a student gives, and when indicated to pass to the next unit. "Super" and "sub" are used to make super or sub-scripts. "Copy", "font/micro", "□" and "edit" are used on the PLATO-IV system only. "Erase" is used to erase the last character typed by the student. The other keys are special function keys:

term/ans
help
lab
data
back
stop

TERM is used in conjunction with the TERM command, and allows a student access to special units from anywhere in a lesson. When the student presses TERM, s/he is asked "what term?" -- an answer of up to seven characters is then checked against a list of TERM commands found in the lesson, and a branch is made to the appropriate unit.

ANS (unshifted TERM key) is used by a student to request the correct answer to a question. If the author of a lesson has programmed the use of ANS, s/he may then show the correct answer, or continue in the lesson.

HELP, LAB and *DATA* function similarly, causing branches to "remedial" or "help" units. An author must specify the name of a unit to be processed in the event a student presses the appropriate key. The unit may change from one base unit to the next. When the student has finished with

the "help" unit (or sequence of units) s/he must return and repeat the base unit.

```
unit      xxx
help      xyx
write     Do you know which TUTOR
          command is used to display
          the value of a student
          variable?
          (Press HELP for a hint)
arrow     601
answer    (show,showa)
no
write     Press HELP for a hint.

...more units...
unit      xyx
at        150
write     Here are some
          TUTOR commands:
           write
           show
           unit
           arrow
end
```

If the student presses the HELP key, s/he is immediately branched to unit xyx, where the hint is displayed. At the completion of that unit, an "end" returns control to the base unit.

An author may activate any combination of HELP, LAB and DATA keys in a unit, and they may initiate branches to any set of units. *It is important that a HELP-like sequence contain an "end" command at its end, to return control to the base unit.*

BACK is usually utilized to either effect a review of previous units, or to get back to an index of materials which may be covered in a lesson. It acts like HELP, LAB and DATA in that it branches the student back to another unit, but it does not require an END, since the unit the student goes back to becomes the new base. It is possible for an author to use a combination of NEXT and BACK to make an easily-reviewed lesson (such that a student can always back up to review a few units).

The *STOP* key can be used to end a lesson. This is done by holding down the SHIFT and pressing STOP. This works on PLATO-IV as well!

Special actions of these function keys are described in the COMMAND SUMMARY section.

NOTE: HYPERTUTOR is usually run on terminals which do not have these special function keys. Therefore, "simulated" keys have been provided. A student may type a "number sign" ("pound sign" to some) and the name of the function key. So s/he will type #HELP rather than pressing the HELP key. Since shifted versions of these keys also exist, shifted function keys are obtained by adding a "1" to the key name. Thus shifted-HELP becomes #HELP1 .

ERASING THE SCREEN

Some of the branching commands (JUMP, NEXT and NEXTNOW) erase the screen when they are performed. At other times an author may wish to erase the whole screen, and this is done with the ERASE command.

In addition, an author often wishes to clear only a few characters, or a small space on the screen. If s/he puts an expression into the tag of an ERASE, the appropriate number of character positions will be immediately erased.

There are several uses of this command. One is to make words "blink" on the screen. If a lesson writes a word on the screen, then immediately erases it, then writes it again and erases, it will seem to blink.

```
unit      blink
write     Stuff...

at        2405
write     Press NEXT
at        2411
erase     4
at        2411
write     NEXT
```

Another nice use is to write a paragraph of information on the screen, then to erase important or critical words and have the student fill them in:

```
unit      xxx
at        503
write     The command which is
          used to show the value
          of a student variable
          is called SHOW.

          Press NEXT when ready.
pause
at        813
erase     5
arrow     813
answer    show
no
write     Perhaps you should read
          more carefully next time.
          Let's try again.
nextnow   xxx
```

(The PAUSE command, introduced here, simply stops and waits for the student to press *any* key, then continues.)

Did you notice that each ERASE was preceded by an AT command to set the screen position?

INHIBITING ERASING

There are times when an author wishes to inhibit the erasing of the screen which goes with JUMP. This can be done with the insertion of:

```
inhibit   erase
```

anywhere before the JUMP. INHIBIT will always inhibit the next automatic erase -- it does not inhibit erases ordered via ERASE commands.

CONDITIONALS

So far, the only conditional operations are those which are answer-contingent. Using information you know so far, you can only make calculation or actions contingent upon the student's answer. In this section you will learn how to make branching (and other actions) contingent upon the values stored in student variables.

Commands are made conditional by changing the form of their tags. A normal tag for JUMP would be a unit name:

```
jump      xxx
```

In the example which will follow, assume that variable "nl" contains the number of wrong answers the student has made during this session. The author wishes to jump to unit "xxx" if the student has made fewer than 10 errors, and to unit "yyy" if the student has made 10 or more errors. The tag of the JUMP then takes on a special two-branch conditional form:

```
jump      nl<10,xxx,yyy
```

condition to be checked — "true" branch — "false" branch

If the condition is "true" the jump is executed to unit "xxx", and if the condition is "false", the jump is to unit "yyy". The same tag could be used with other commands:

```
join      nl<10,xxx,yyy
next      nl<10,xxx,yyy
```

One can think of HYPERTUTOR evaluating the expression, choosing the appropriate name from the list and then creating a regular command, such as:

```
jump      yyy
```

which is executed.

A second type of conditional exists, in which names are listed, and a value is chosen from the list according to the *numeric* value of the expression:

```
jump      nl,x,xxx,xxx,xxx,xxx,yyy,yyy
```

expression — "-" "0" "1" "2" "3" "4" "≥5" — branches

The actual list may be of varying length. The first unit name in the list will be chosen if the expression is negative, the second will be chosen if the expression is equal to zero, the third if it is equal to one, etc. The last item in the list is always chosen for values greater than those

in the list. For example, if "nl" is equal to 3, unit "xxx" would be the target of the JUMP. If "nl" is 4, "yyy" becomes the target. And if "nl" is 5 or greater, "yyy" is also the target. A special unit name, "x" has been inserted in the negative spot in the list. If unit "x" is chosen as the result of the operation, the command is simply *not performed*. Unit "x" may be used anywhere in a conditional list, and the command is always skipped when "x" would be the result. (Note: It is therefore impossible to ever use "x" as a unit name, since one cannot branch to that unit in any way.)

Conditional lists can be used on many commands. A few are listed here which may be most useful: NEXT, BACK, HELP, NEXTNOW, JUMP, JOIN, GOTO, LAB, DATA, BASE and FINISH.

The special "unit name" "q" is also available for use in some conditional forms: NEXT, BACK, HELP, NEXTNOW, GOTO, LAB, DATA, BASE and FINISH. When "q" is the result of a conditional, the unit name is *cleared* for the command -- a :

```
next      q
```

would clear the NEXT unit pointer so that it could either be reset by a subsequent NEXT or so that the next physical unit would be used.

```
base      q
```

would clear the base unit pointer so the current unit would become the base.

```
goto      q
```

would cause termination of the current unit (that's what GOTO does) and then control would go to the appropriate next base unit, since to GOTO unit is specified other than "q".

SPECIAL HYPERTUTOR VARIABLES

Authors working on a system where there are many types of devices need additional information when a lesson when a lesson is running. A number of special variables can be used within a lesson -- some of these correspond to PLATO-IV variables and others are unique to HYPERTUTOR.

Standard TUTOR-IV special variables:

jcount	ntimes
error	wcount
where	lcommon
wherex	baseu
wherey	proctim
	router

HYPERTUTOR additional special variables:

- lesson
- cost
- oldtime
- nextname
- oldcost
- disk
- time
- name

jcount -- equals the number of six-bit characters in the last student response.
error -- = -1 if there was an error on the last operation
where -- equals the gross-coordinates of the current screen position.
wherex -- equals the fine-grid "x" coordinates of the current screen position.
wherey -- equals the fine-grid "y" position
ntimes -- equals the number of times the student has attempted the current arrow.
wcount -- the number of words in the student's answer. Set only after a MATCH, ANSWER or CONCEPT has been processed.
lcommon -- length of common, if loaded.
baseu -- name of current base unit.
proctim -- cpu time in seconds.
router -- name of router lesson.

lesson -- name of the current lesson.
cost -- running job cost in dollars.
oldtime -- seconds spent in last lesson.
nextname -- name of the "NEXT" key for this particular type of terminal.
oldcost -- cost of the last lesson run.
disk -- number of disk accesses made by this lesson, upper 30 bits=ECS accesses.

time -- time since sign-on.
name -- student's sign-on name.

These special variables may be accessed from any CALC or expression.

INDIRECT REFERENCES

HYPERTUTOR variables are normally referenced by format and number, but it is possible to reference them indirectly as well. This is done by indicating a reference format, followed by an expression, within parentheses, which (when evaluated) gives the actual student variable number to be accessed. The simplest form of such a reference would be:

```
calc      n(1) = 7
```

the "1" is an expression, giving a value, and "n1" would be referenced. More complicated expressions:

```
calc      n(n7) = n7
calc      n(n44+1-n3) = n(8+n2)
```

can also be used. The expression within the parentheses is always evaluated first, then used as the student variable number.

A pertinent example is a unit which would set the first ten variables to zero:

```
unit      xxx
do        yyy,n99 = 1, 10
jump      zzz
unit      yyy
calc      n(n99) = 0
unit      zzz
...
```

(DO is a command which repeatedly executes the named unit, "yyy", first setting n99=1, then stepping it one number at a time until it reaches the termination value, "10".) The unit "yyy" gets executed ten times, setting the first ten student variables to zero, corresponding to values of n99.

Indirect references are valid in all locations where direct references are valid.

GRAPHICS

Most of the CRT and printing terminals in use on HYPERTUTOR systems will not have the complex graphics capabilities of the PLATO-IV terminal. Readers who do not have PLATO-IV terminals may skip this section entirely.

The simplest graphics available are the line-drawing commands. TUTOR-IV uses a single DRAW command encompassing both fine-grid and gross-grid systems. HYPERTUTOR uses separate commands, DRAW and DRAWF, which can be directly translated to DRAW commands when transferred to a PLATO system.

DRAW specifies gross-grid coordinates and HYPERTUTOR connects these coordinates with single lines:

```
draw      101,164,3264,3201,101
```

This example would cause the drawing of a line from line 1, character 1 to line 1, character 64, then to line 32, character 64, and so forth. An author may specify that HYPERTUTOR is to skip to a new location before continuing the line by inserting "skip" in place of one of the coordinates:

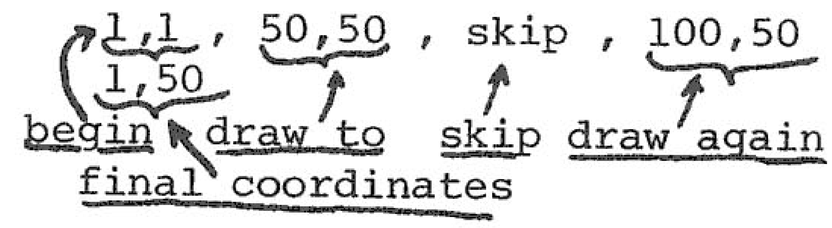
```
draw      101,164,skip,3264,3201
```

The example above would draw a line from line 1, character 1 to line 1 character 64, then skip to line 32, character 64, where it would then draw to line 32, character 1.

> *Note: TUTOR-IV on the PLATO-IV system uses semi-colons (;) to separate coordinates in the gross-grid version of DRAW. Semi-colons will be accepted if found in a DRAW list.*

DRAWF is the HYPERTUTOR fine-grid version of DRAW. "x" and "y" coordinates are specified in pairs, and "skip" may also be used:

```
drawf     1,1 , 50,50 , skip , 100,50
```



The DRAW command is simulated on character-oriented CRT terminals. A dotted-line will be drawn to approximate positions on the screen. The DRAWF command will only work on PLATO-IV terminals.

The MODE command can be used to change the drawing mode of the PLATO-IV terminal. Normally, the system is in WRITE mode, in which dots are lit to represent characters, and lines are drawn, without any change in background information already on the screen. In the REWRITE mode, however, the background of each character is erased before the character is written on the screen, ensuring readability. And in mode ERASE, dots are unlit (or erased) rather than being lit.

The terminal mode may be changed by a MODE command within the TUTOR text of a lesson. For example:

```
unit        xxx
draw        101,110,510,501,101
mode        erase
draw        101,110,510,501,101
...
```

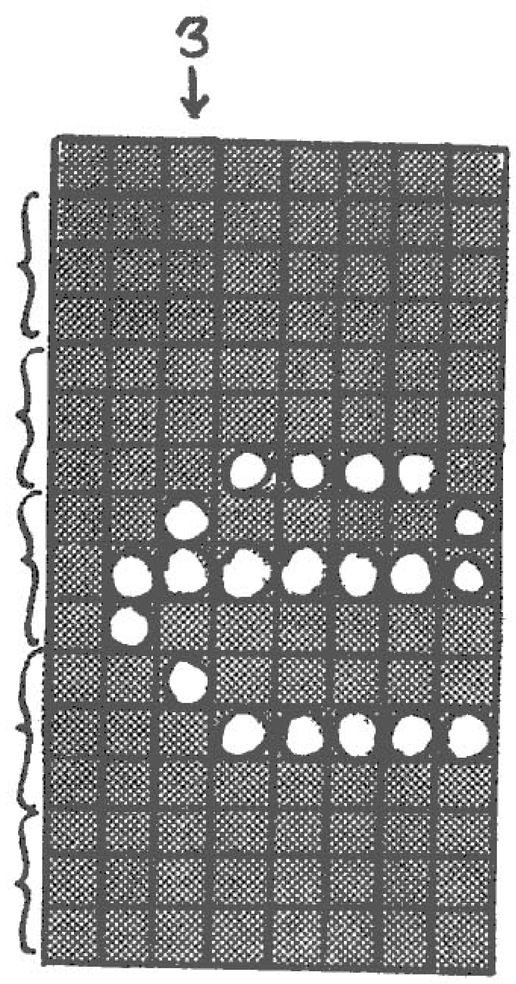
would cause the drawing of a small box on the screen, followed by its erasure, one line at a time. Any commands which follow would also appear in ERASE mode, unless the author changes back.

The MODE command has no effect on regular CRT terminals.

SPECIAL CHARACTERS

The CHAR and PLOT commands combine on the PLATO-IV terminal to provide a wide range of character sets and specially drawn figures. Special "lessons" exist on PLATO-IV to help design these special characters, but it helps to plan the characters ahead, and the following section is a full discussion of how these characters may be manually created and inserted into TUTOR lessons.

A character-position on the PLATO-IV terminal may be thought of as an eight by sixteen-dot grid on the screen. Characters are formed by lighting the appropriate dots from the grid -- this is the technique used on scoreboards and moving light displays. Here's the grid with a lower case "e":



The grid can be thought of as a series of eight columns, each containing sixteen dots. In the binary code used by the computer, each dot's on or off status could be indicated by a "1" or "0". Thus, the column in the "e" marked as column 3 would be represented as "0000000110100000". These binary numbers can be converted to "octal" representations by taking them three digits at a time (from the right end) and using the following equivalents: 000 binary = 0 octal, 001 = 1, 010 = 2, 011 = 3, 100 = 4, 101 = 5, 110 = 6, 111 = 7. Because the leftmost digit of the binary number is a loner, it will always be converted to 0 or 1 octal. The string from column 3, above, would be "000640" octal. An "o" (oh) is prefaced to indicate that a

number is octal in TUTOR, so the correct representation for column 3 of the letter "e" is "o000640". The whole letter is created by listing the columns from left to right, in a CHAR command:

```
define    e = 5
char      e
          o000000,o000300,o000640
          o001220,o001220,o001220
          o001220,o000620
```

As an author gets better at recognizing these octal patterns, s/he often disposes with the binary representations and simply codes directly from a picture of the letter into octal.

To make small creatures, one can combine several characters. The grids on the next page can be duplicated and used as workspaces for such creations. At the top of the page is a set of six spaces, each one-character in size, positioned for larger drawings. To create a rabbit, for instance, you may wish to sketch an outline on the grid, then fill in dots until you're satisfied, and finally create the octal representations.

In the example above, and in most TUTOR programs, a special name has been defined for the character. It is listed in a DEFINE command, then used in the CHAR -- each character is given a unique number (between 1 and 128) which tells the terminal where to store it in memory. In the example, "e" will be stored in position five.

To plot a character on the screen, one first determines the position by using an AT command (or from the position of the last WRITE or PLOT), then writes a PLOT command indicating the character number to be plotted.

```
plot     e
```

...would plot character 5, "e". We could also have used:

```
plot     5
```

...with the same result, but this form is more awkward, obviously.

Plotting characters made up of more than one of these small characters involves setting a screen position for the upper row, sending PLOT commands for the characters in the top row, then resetting the screen position with another AT and plotting subsequent rows.

Creatures made up of such characters can be made to "move" by plotting them, then (in mode REWRITE) plotting them again a dot or two to the right or left. The plotting takes place with such speed that there is an amazing illusion of movement.

It is also possible to give the illusion of movement by plotting, erasing, replotting at a nearby position, erasing, etc.

It is important to realize that the CHAR command is actually executed when it is encountered in the lesson. Because each CHAR load takes about 1/10 second, it is wise to stagger these loads throughout a lesson. One might load five or ten characters before displaying an initial welcoming message, then five or ten more, and perhaps load special characters for animations just before they are needed. Because the PLATO-IV terminal's character memory can be reloaded at any time, one can discard unneeded characters by reusing their spots in the terminal memory.

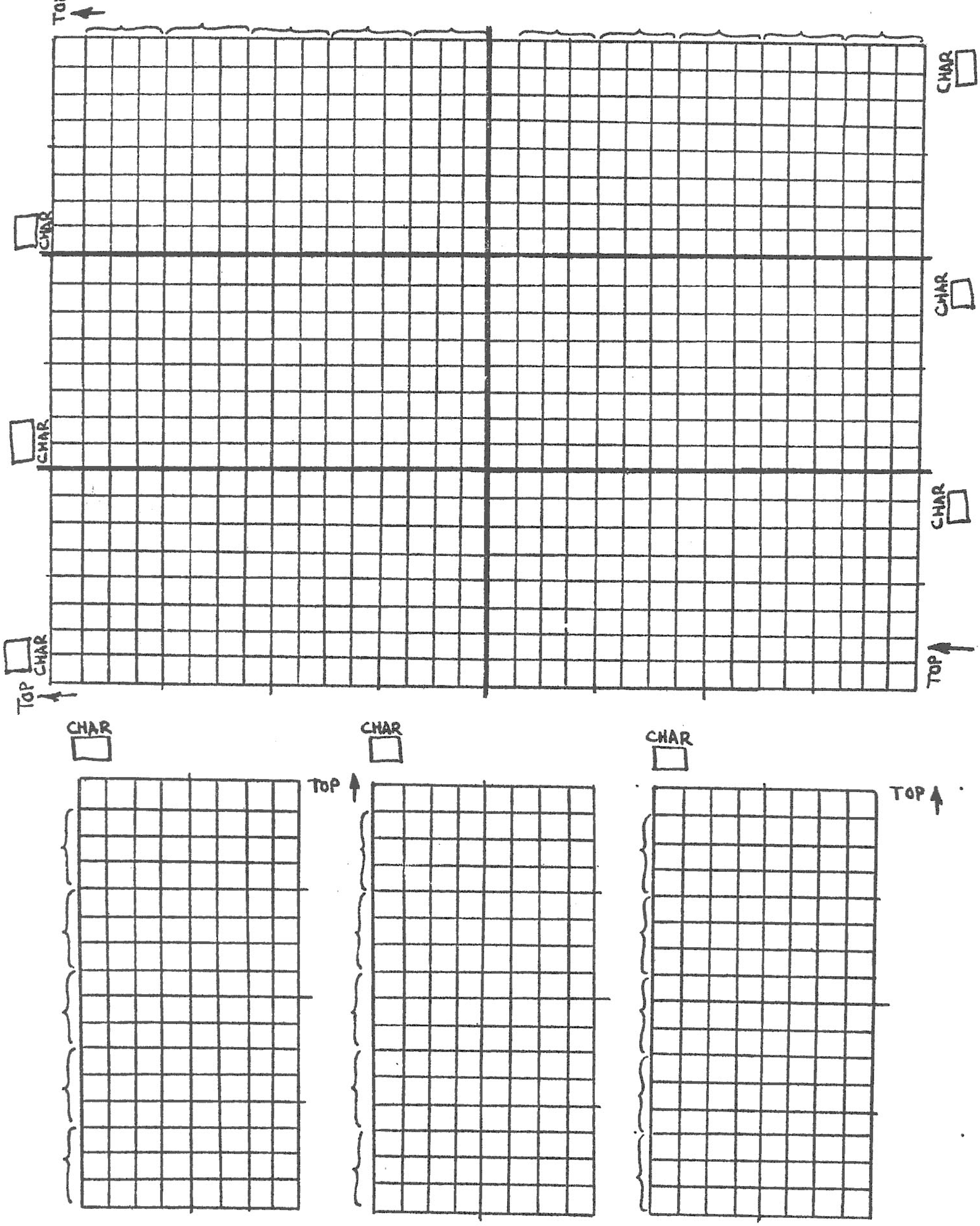
WRITING SIZES

Though not implemented in HYPERTUTOR at this time, on the PLATO-IV system it is possible to write in many sizes by specifying a multiplier in the SIZE command:

```
size     4
```

The multiplier is applied to subsequent WRITE or SHOW commands, and all figures appear that many times normal size. To do this, PLATO enters line-drawing mode and creates the letters and numbers from line segments, which is slower than the normal mode. Normal writing may be resumed by executing a "size 0" command. HYPERTUTOR will have this ability some time in the next year.

(37)


TOP
CHAR
CHAR
CHAR
CHAR
CHAR
CHAR
CHAR
TOP
TOP
CHAR
TOP
CHAR
TOP
CHAR
TOP

One of the more perplexing problems associated with HYPERTUTOR programming is the potpourri of terminals available.

SOME WEIRDO TERMINALS

Notes: 1) To run on all "CRT" terminals (including PLATO-IV), a screen size of 24 lines by 64 characters must be observed. This is the *largest* page which will fit on all devices listed below.

2) The system will change the message printed by the NEXT and NEXTNOW commands -- the programmer is responsible at all other times. (This is because the CR or NEXT key changes name from terminal to terminal.) The safest way to tell what to call that key is to check the value of TERMTYP, a student variable. [This is important -- most HYPERTUTOR users are extremely naïve and will be very confused if you ask them to press a key they can't find on their keyboard.]

3) On printing terminals, things will be written in the physical order your commands are processed. AT commands will cause carriage returns <u>if</u> they would be printed on <u>different</u> lines on a CRT terminal.

4) Many terminals have a full-line erase capability. Using this feature, the cursor may be positioned on a particular line and the whole line instantaneously erased. On the Hazeltine-2000, this will erase the <u>full</u> line, no matter where the cursor is positioned. On ADDS 580 or 880 the line is erased from the cursor position to the end of the line. The PLATO-IV terminal has no line-erase, and therefore, a full line erase is done by sending 64 blanks. This takes quite a bit longer than the other terminals would.

5) There is a rather general heirarchy of terminals, beginning with printing terminals (TI 700, Teletype 33, etc.) which cannot backspace or move upward on the paper. More complex are the CRT terminals without graphics (Hazeltine 2000, ADDS 580 and 880) -- the AT command will work on these, but not on printing terminals. And finally, the "graphics" terminals (the PLATO-IV), which can draw lines or special characters.

	Teletype model 33	Texas Instruments 700 series	Hazeltine 2000	ADDS 580	ADDS 880	PLATO-IV
Printer	X	X	X			
CRT			X	X	X	X
Speed(char/sec)	10	10/30	10/30	10/30	10/30	120
Lines			27	24	24	32
Chars/line	72	80	72	80	80	64
CR key is	return	car ret	cr	new line	new line	next
TERMTYP =	0	0	-1	-4	-5	-6
Draws lines						X
AT works			X	X	X	X
Special features						microfiche touch-panel

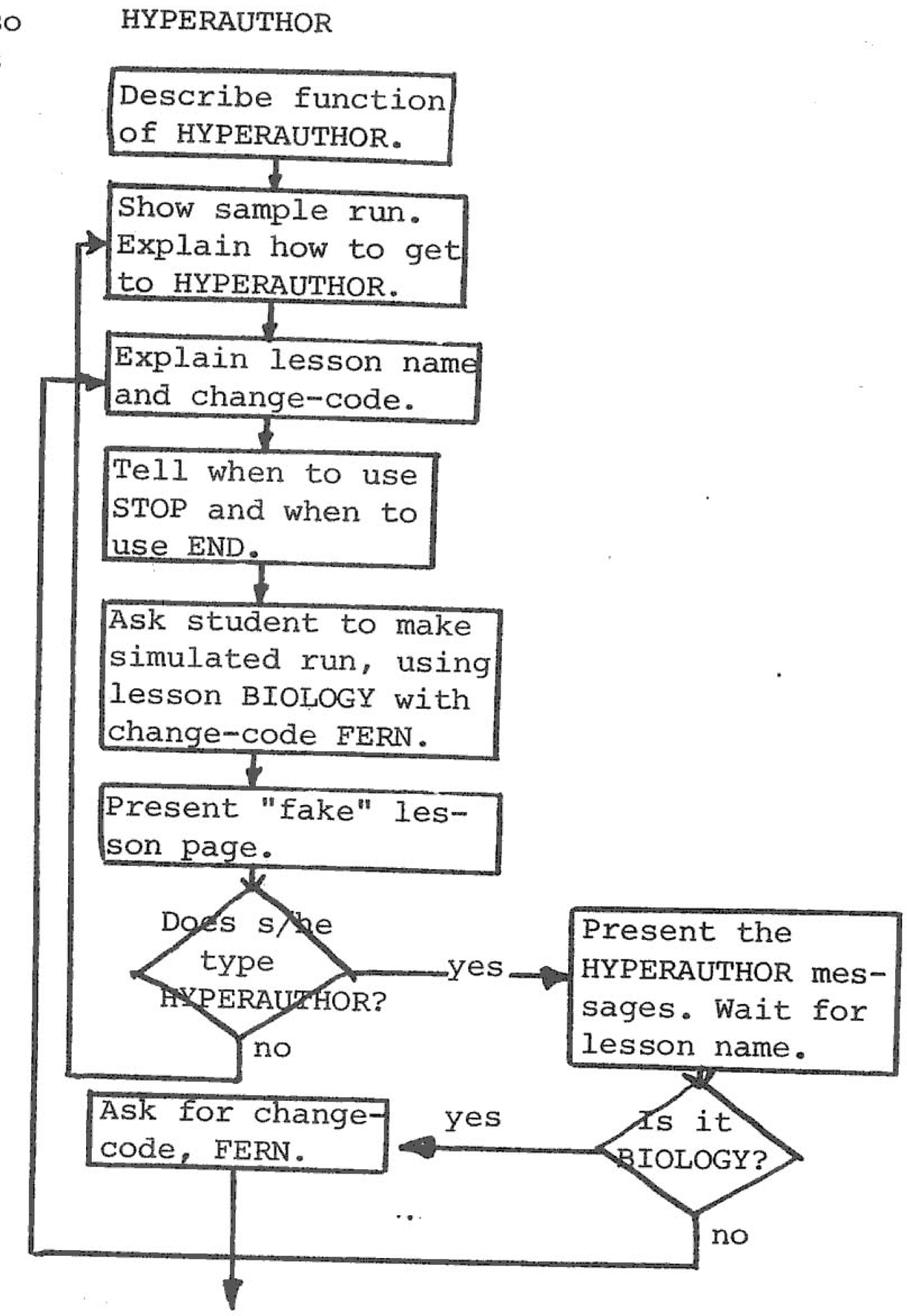
DOCUMENTING IT

A lesson must be properly documented so that potential users can determine its usefulness. This documentation takes several forms, and much of it can be prepared before you even approach the computer to enter TUTOR commands.

The first step is to specify exactly what your lesson will cover, in terms of the subject matter. By "exactly" we mean that you should write a short paragraph or abstract, intelligible to someone with average knowledge of the subject area, which specifies what kinds of materials will be presented, and how you will determine that the student has in fact mastered the materials. If you know about setting behavioral objectives, this is a good way to start a lesson.

Then, it is recommended that you draw a flow-chart which describes the basic "flow" of students through the lesson. The flowchart need not be terribly explicit about how the program will perform "internally", but it should be intelligible to someone who has read your overall objectives.

Below are objectives for part of a short lesson. To the right is part of a flow-chart prepared for that part of the lesson dealing with the HYPERAUTHOR:



Overall description: This lesson introduces a potential author of TUTOR lessons to the special author-aid lessons available on HYPERTUTOR.

Lesson introduced:	Objective:
HYPERAUTHOR	User will be able to specify a lesson to be edited (as for HYPERAUTHOR) and will be able to use STOP and END commands in appropriate conditions.
HYPERTEXT	User will successfully specify NAME, TITLE, AUTHOR, LEVEL, STATUS, SUBJECT, UP, DOWN, RIGHT and LEFT commands for a test lesson.
PRINT	User will send a listing to be printed at appropriate terminals and central site.
	User will document using *, ** and *** and

Flow-charting is not difficult, and a few simple rules will help:

1. Don't put too much into a single box -- usually a box corresponds to a unit (or at most two or three units).
2. Put enough description in that a novice can determine what you're talking about, perhaps with reference to the overall description and objectives. You don't need to show what the student is going to see, but it helps to include a few sample screenfuls in your documentation.
3. Diamond shapes indicate decisions to be made by the program. They are the *only* boxes with more than a single arrow coming out of them. Each exiting arrow *must* be marked to indicate how it corresponds to the answer to the question within the diamond box.
4. Since flow-charts are supposed to clarify the flow within a lesson, you should make them simple! Don't try to impress anyone with the complexity of your lesson. If necessary, make an "overall" flow chart which shows the lesson at a gross level, and several more-detailed charts for sub-sections of the lesson. [The example on the last page was only a small part of a rather large lesson.]

You will find at some point in your programming career that flow-charting becomes an onorous task which you believe to be quite unnecessary -- don't give up! It is important that prospective users be able to decipher your lesson and a graphic presentation is often worth far more than words. And because lessons always grow far beyond the size you project for them, a flow-chart is often the only roadmap you will understand after the lesson is a few months old.

At this point it is wise to begin thinking about how the C.A.I. lesson is to be implemented. One must make a choice of terminal -- if the lesson requires complex graphics, animations or fast interaction, one normally chooses PLATO and its versatile system. If there is a lot of presentation of text, with little interaction, one often can decide *against C.A.I.*, and not even bother with programming. If the lesson is a simulation, requiring much computer time, one will pick the local system with the greatest amount of free time. CRT terminals will be used in lessons which require only limited graphics. A printing terminal will be used when the student requires a printed record of the interaction. If slides are required, a microfiche may be made for the PLATO terminal, or it may be cheaper, faster and easier to simply make line-drawings on paper, duplicate them and distribute them to the students in class. If you need data on how students are performing in the lesson, you may have to arrange in advance for storage space, and at any rate you will have to plan the type and extent of data to be recorded.

INTERNAL DOCUMENTATION

All programs (or lessons) must have sufficient "internal" documentation that a programmer can figure out what's happening. In TUTOR, one inserts remark lines by placing an asterisk in the first column, and then placing the comment in the tag. These lines can be used to insert a *variable map* at the beginning -- the map indicates what each of the student variables is used for. In each unit where a decision is made, comments are used to indicate the basis for the decision and the actions taken. Where calculations are needed, comments explain the general tenor of these operations, perhaps giving background equations or information. This may seem like a lot of work, but an adequately commented program is much easier to understand than one in which you have to figure out every command.

(41)

CONDUIT DOCUMENTATION GUIDELINE

Abstracted Version

PRELIMINARY VERSION

This is a preliminary version, not intended as a final copy. We would appreciate any comments.

CONDUIT: A consortium of regional networks at Oregon State University, North Carolina Educational Computing Service, Dartmouth College and the Universities of Iowa and Texas (Austin).

This guideline is being developed to apply to all sorts of educational computer materials, not just C.A.I., but it supplies valuable information to the C.A.I. author as well.

INTRODUCTION

The CONDUIT Documentation Guidelines describe standards for packaging computer-based curriculum materials. These guidelines stipulate those types of documentation deemed necessary not only for local usage but also for transportation of packages, both educationally and technically, among multiple non-homogeneous computer installations. Documentation to comprise a complete package will be collected from appropriate sources serving the following purposes:

DOCUMENTATION CHART

DOCUMENT	*SOURCE*	*TARGET*	*PURPOSE*
1. COVER LETTER	Local Curriculum Coordinator*	Faculty User	Availability announcement
2. ABSTRACT	Author	Reviewer or Discipline Committee: Faculty User	Preview
3. EDUCATIONAL DOCUMENTATION	Author	Reviewer or Discipline Committee: Faculty User, Student	Substantive & Pedagogical Aspects
4. TECHNICAL DOCUMENTATION	Programmer, Author	Reviewer or Discipline Committee: Curriculum Coordinator Faculty User, Student	Technical Aspects (systems & programming considerations)
5. LOCAL DOCUMENTATION (educational & technical)	Local Curriculum Coordinator	Faculty User, Student	Modifications resulting from usage at a specific network computer center
6. REVIEWS	Reviewers, Discipline Committee**	Faculty Users	Certification

NOTE: This abstracted version was prepared by Trinka Dunnagan from the *CONDUIT Documentation Guidelines* of October, 1973 in which each component of documentation is explained more fully.

*Each local Curriculum Coordinator is responsible to CONDUIT for collecting and disseminating computer-related curriculum material at his network schools.

**Discipline committees, who serve in an advisory capacity to CONDUIT, are composed of faculty with interest and expertise in computer applications within a discipline.

1. COVER LETTER

A *cover letter* should be supplied by the local Curriculum Coordinator as supplementary, local documentation for any CONDUIT package. The purpose of such a letter is to provide prospective network users with a realistic picture of their relationship with CONDUIT in the event of the package's adoption and to state the costs, availability, and procedures for acquiring any materials (texts, card decks, magnetic tapes, program listings, etc.) needed for that adoption. Any discussion of availability should list in addition to the source repository other network centers and computer types at which this package is supported. Any restrictions (such as copyrights, etc.) on the reproduction and use of these materials should be stated at this time as well as the caveat that although CONDUIT has attempted to maximize the educational value and technical reliability of the programs, *no liability* for errors in these materials, expressed or implied, is assumed by the author(s), the source facility (NAME), or the CONDUIT consortium. User's expectations of CONDUIT support (computer time, programming aid, consulting services, etc.) should be specifically covered in terms of extent, personnel and mechanics involved.

2. ABSTRACT

The author of each package should provide an abstract sufficient to serve as an all purpose introduction to the computer-based curriculum package. The abstract should contain the following items:

(a) Descriptive title
(b) Mnemonic or calling name
(c) Author(s) and original source
(d) Names and locations of any subsequent modifiers
(e) Summary of substantive content
(f) Statement of educational objectives
(g) Specifications of computer's instructional role
(h) Background requirements for instructors and students
(i) References (if available)

3. EDUCATIONAL DOCUMENTATION

The author of a computer-based curriculum package addresses in this type of documentation the needs of faculty and student users. Topics covered should include the following:

(a) Substantive aspects—theoretical background and disciplinary principles of this computer-based package.
(b) Pedagogical significance—relationship of the educational objectives stated in this material to content and instructional techniques.
(c) Implementation considerations—suggestions for instructional management within the classroom, the computing environment and the standard curriculum.

These topics could appear in separate single-purpose manuals (e.g., Teacher's Guide, Student Manual, Programmer's Guide, Problems, etc.) or as a collective write-up depending on the size and scope of the package being documented.

4. TECHNICAL DOCUMENTATION

Technical documentation involves those materials necessary for the understanding by a potential user of how the courseware (e.g., program) operates. Technical documentation should attempt to extricate that programming documentation which is universal and transportable, in terms of information content, from site-specific descriptions needed only for local usage. In order to clarify both the program logic and function, one needs:

(a) Explanation of the program's logical organization and the functions of any discrete modules.
(b) Well-commented program listing.
(c) Descriptions of sample input such as test data and any program parameter cards required.
(d) Listing of output generated by such sample input.
(e) Test problems and expected results.
(f) Description of program options and how to exercise them.
(g) Listing and explanation of program-generated messages.
(h) Glossary of variable names and special discipline oriented items.
(i) Itemization and descriptions of supporting programs, subroutines and external files.
(j) Formats for parameter and data cards.

For each data set, the following information should be available:

(a) Description of file organization.
(b) File data item definitions.
(c) Indices to the data set.
(d) Security procedures, if any.

5. LOCAL DOCUMENTATION

Curriculum Coordinators should provide network users with instructions for usage of CONDUIT materials at their local computer center. Information classifiable as local (or site-dependent) is necessary only if it differs from that which is provided as universal documentation. Such information includes:

(a) Instructions and examples on how to use the local computer (JCL, deck setup, typical file or program access, major operations.
(b) A setup example of how to access the program under consideration.
(c) A listing of the local version of the program and notations on its significant differences from the universal model.
(d) Sample I/O for this version of the program.
(e) Information on typical run times and related costs.

In addition an overall description in explicit terms of the current computing environment should state:

(a) Operating system, release level/version.
(b) Mode of usage (batch versus interactive).
(c) Storage devices required.
(d) Special peripheral device requirements (central site).
(e) Special terminal needs (user site).
(f) Common causes of program failure and error recovery.
(g) Any other locally pertinent information or comments on these materials.

6. REVIEWS

Material selected for dissemination through CONDUIT is first reviewed under the direction of an appropriate discipline committee. Those packages receiving positive reviews are included in the CONDUIT Library for undergraduate education with solicited reviews comprising a final section of the documentations.

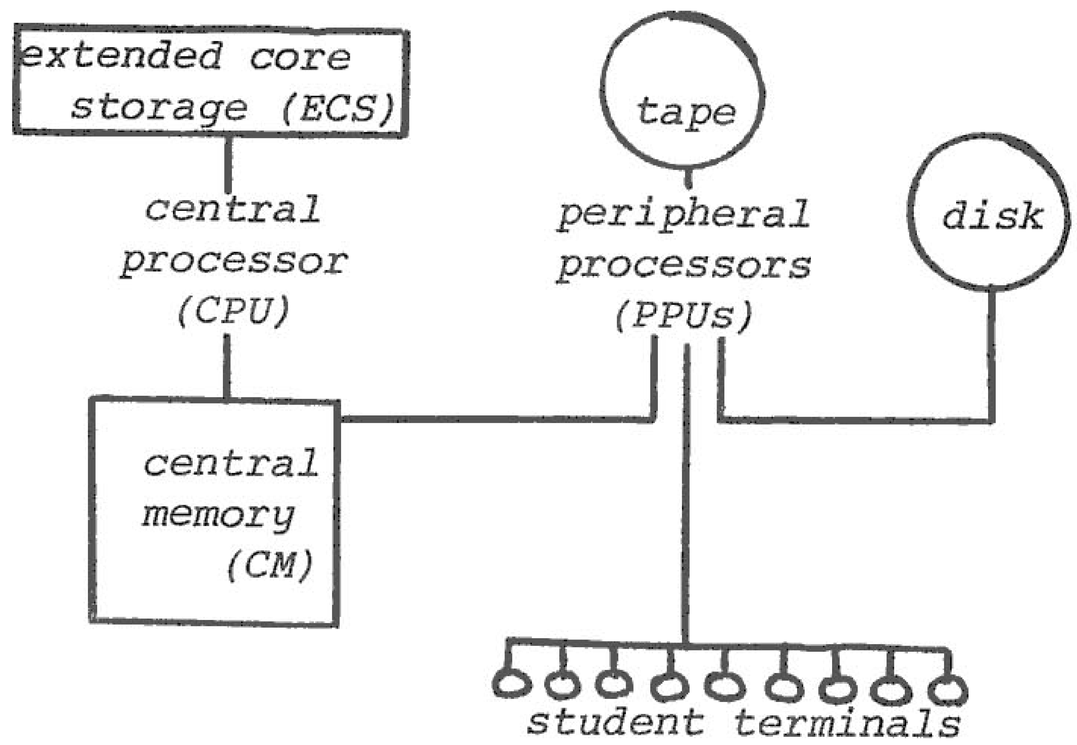
SPECIAL "LESSONS"

Things to be typed by the computer-user are italicised in this chapter!

Special "lessons" are required to do the dirty work of maintaining the C.A.I. system, and some of these are described in this chapter.

RESTORING "IDLED" LESSON TEXT

The diagram below shows the construction of the CDC 6400 computer. Lesson text and student versions (condensed) in the lesson directory are all stored on the *disk storage units*. However, because storage space is limited, these files of information cannot be maintained indefinitely. Those which are not accessed at least every five days are semi-permanently moved (or "idled") onto a set of magnetic tapes, *the idle tapes*. This is done once each day, before the computer is readied for student use, at the direction of the computer operator.



Lessons stored in the lesson directory are protected from this process, but the TUTOR text of a lesson is not. An author who does not HYPERAUTHOR his lesson at least every five days will find that the lesson has in fact been idled, and must be restored from one of the "idle" tapes. The next paragraph outlines this procedure, and is followed by an explicit example.

1. The lesson-author must determine which of the idle-tapes holds the text.
2. S/he must request the appropriate tape of the computer operator.
3. The computer must be told which file of text to retrieve, and then told to get it.
4. The file of text must be read into the Editor program again and catalogued as a semi-permanent file.
5. The tape is then returned for use by other people.

1 -- The author first enters the regular HYPERTUTOR system and requests the lesson "EDDIE", which puts him into the standard Northwestern Editor program. Then, the author types:

$userlib,pull,tell,xx,idlefiles.

The initial "$" indicates that this is a control-card (part of the job-control language of the 6400 computer). It activates a program named *userlib*, which *pulls* a record from a file named *tell*, and places it on a local file *xx*. The record to be pulled is *idlefiles*. The message:

successful execution

will be printed when userlib has finished its task. If this message isn't printed, it is appropriate to contact the computer center, after checking to be sure you've typed everything correctly. The listing of idle files is now available on a local file "xx", which must be read into Editor:

read(xx)

When you've typed the line above, the Editor will respond with the number of lines read (equal to the number of idle files on record):

1907 lines read

for example. The computer can then be ordered to search for the text file of a lesson. All file names begin in column 41 of the listing, so Editor is told to search beginning in column 41, for the exact name of your text file -- the exact name is created from the lesson name you give HYPERAUTHOR by adding

asterisks until you reach a total of ten characters. (A four-character name would have six asterisks added, a seven-character name would have three additional asterisks.) For example, a lesson called "HYPER" could be found by entering:

*(41)search"hyper*****"*

If the lesson text has been idled several times, the Editor will find several lines of the following form:

```
11920 dump  31  idL004 12/17/73 E  0 C  1  hyper*****
11930 dump 257  idL021 11/15/73 E  0 C  1  hyper*****
```

If none are displayed, it is possible that the lesson text is not on any of the idle-tapes, and will have to be restored from a user-owned tape. It is *always* wise to maintain copies of lesson text on your own tape. Instructions for doing this are contained in the next section of this chapter. If several lines are displayed (as above), one should examine the dates to determine which is the latest version. In most cases, this will be the first line -- it was created December 17, 1973 in the example. Directly to the left of the date is the tape number, "idl004" in this case. It pays to write this number on a slip of paper for later use.

Now, the author proceeds to the actual retrieval. First, because the Editor contains the idlefile listing, it must be cleared:

clear

The computer responds:

cleared

The file will be retrieved by a program named *getfile*. Getfile reads instructions from input created in the Editor. This input contains a) the local name to be assigned to the retrieved file, b) the type of file, c) the name of the file, d) its cycle, and e) its passwords. For example:

*1.a,pf,hyper*****,1,mmb,mmb.*

The doubly-underlined portions are the parts you have to worry about -- everything else can be typed exactly as indicated. The first variable portion is the filename, exactly as created for the searching procedure (by adding asterisks until ten characters are reached). Then, repeated twice, the change-code (or password) for the lesson text. Be sure you include the terminal period. Then this information must be written to the input file, so getfile can find it:

rewind(input) ready input for writing
cwrite(input) write text, no numbers
rewind(input) ready input for getfile

Having gone through this rather strange procedure, the author must tell the computer operator to get the tape. This is done by another control-card:

you fill in

$LabeL,syschec,L=systemcheckpoint,vsn=idL004.

In essence this sends a message to another human being who mans the computer. Because this takes some time (searching for the tape, putting it on the computer, etc.), one normally expects at least a five-minute wait at this point. When the tape is ready the computer responds:

successful execution

The tape is ready and the computer is ready. Getfile is then called:

$(22000)getfile.

Again, there will be a wait while the computer searches the idle-tape. The length of the wait corresponds roughly to the number following the word "dump" in the idlefiles listing. In the example this is "31", meaning that our file is the thirty-first file on the tape. It will take only a short while, therefore. This process may take up to fifteen minutes during heavy periods. After another:

successful execution

the file is locally available to the Editor. It is read into working storage:

accept(a)

and re-catalogued:

new=hyper,pw=mmb

name change-code

If the "accept" was successful, the computer will list the number of lines read. At any rate, the file is then available for inspection, and it is always wise to list the first few lines:

list,10,100.

just to be sure you've gotten to the right text file.

To return the tape for use by others, type:

return(syschec)
end

The Editor will terminate, with your lesson's TUTOR text fully restored. Because this does not create a new lesson-directory copy, it is necessary to enter HYPERAUTHOR, give the lesson name and change-code, then type STOP, to condense a new copy.

Note of caution: Editor files (the TUTOR text of a lesson) are actually only guaranteed to exist ten days before expiring, even though they may be found on idle-tapes. It is always wise to keep separate copies of the text, using the SAVE and RESTORE lessons described below.

SAVE AND RESTORE

Magnetic tapes are untrustworthy fellows, and it's always wise to have more than one copy of your lesson text around, in case a tape fails. To make the writing of tapes easy, two lessons, SAVE and RESTORE, have been part of the repertoire of the C.A.I. system for several years. These "lessons" ask you for information about your lesson and about a tape you must buy and store at the computing center, then create a "batch" job which will save or restore your text at a later time. When you are saving text on a tape, you may also write a separate copy of the tape, for safekeeping.

There are a few primitive safety factors built into SAVE and RESTORE, allowing you to intercept errors and recover information which might otherwise be lost.

The tape -- The first step is to buy two magnetic tapes and store them in the computing center. You can easily store 25 to 50 small TUTOR lessons on a 600 foot tape, and more on a 1200 foot tape. When you deposit the tapes at the computing center, you will be given a tape number and asked to pick two additional letters to go along with the number -- for illustration, let us assume you pick "zt" for the letters and that tape number 5000 has been assigned as your main tape, with 5001 as the backup.

Saving -- Once in HYPERTUTOR, ask for lesson SAVE. It will display some initial information (which you should read, of course) and ask for the editor-file name and password. The name is the lesson-name, and the password is the change-code you assigned when you wrote the lesson. Assume you have a lesson "hyper" as before, with change-code "mmb". You would type:

hyper,pw=mmb

It is important that you insert the comma and equal sign exactly as written above, without extra spaces or periods. This line of characters is used to identify your lesson in the directory which will be written on the tape, and cannot be changed. (If you change the change-code on your lesson, it will be saved or restored as an entirely different lesson.) SAVE will link to the Editor, which will get the lesson and write a local copy. If it is successful, it will list the number of lines read -- if unsuccessful, error messages will appear. In the event of error, it is best to type:

%drop

to terminate the SAVE lesson, then try again. If all is ok, you will press the NEXT (or "return" key) to continue in SAVE. You then are asked your charge-number, which you type in its standard form -- two letters, four numbers, a dash and four more numbers. Use no periods, extra spaces or commas. Often a user makes the mistake of inserting "CH" before a charge number, because of experience with the "batch" system. Mistakes such as this are caught by SAVE, and you are asked to repeat. SAVE then asks for your initials.

The initials (or your name) are important, because they identify your job to the owner of your charge-number if you are in a class. In fact, class numbers will not run without the initials. Following that, SAVE asks for the tape number -- this must be four digits and two letters:

5000zt

would be the proper answer for this example. And SAVE asks for any further instructions -- you answer "yes" if you are making the initial run on your tape or if you need a safety copy made. Because tape 5000zt is new, we must answer "yes". We then tell SAVE we have a new tape, which requires special actions. A special label is written on the tape to protect it against accidental destruction. (This label doesn't protect against malicious damage or against any careless actions you may take when examining the tape manually, of course.) And SAVE now asks for your name and phone number -- this information is kept with the batch job that handles the actual tape. It is used in cases of tape failure or operator mis-handling of the job -- the operator is asked to call you if something goes wrong. Because most users cannot deal directly with such errors, we suggest that you call the C.A.T. project immediately in such cases so that we can handle the problem.

SAVE always terminates by telling you that a "batch" job has been submitted to copy your file to a tape. It will give the job name on the screen, which you may copy for later reference. The "batch" job does not instantly run, and may in fact take several hours to complete. Thus, you may find records of such jobs in your monthly billing, or you may notice your account balance changing, at any time up to several hours after a SAVE run. *Note: Because SAVE has no information on whether the batch job was successful, you must pick up the printout for the batch job at the computing center in order to be absolutely sure it ran correctly. Under normal circumstances, however, users will not need to pick up outputs.*

Restoring -- Saving a lesson file on tape is really the critical and difficult phase. Restoring is simple by comparison. One enters lesson RESTORE from the HYPERTUTOR and is given a short explanation, then asked to type the filename and password. They must be typed *exactly as typed when SAVEd*. The characters are used to search the directory on the tape before restoration is performed. RESTORE asks for charge-number, initials and tape number exactly as SAVE did, and they should be entered in exactly the same form. Immediately after you enter the tape number, RESTORE will terminate -- a "batch" job has been submitted.

Restoration also requires operator action, which may require fifteen minutes or more at heavy periods. If an author has nothing else to do, we recommend entering HYPERAUTHOR and requesting the lesson which is being restored. If it is not yet restored, HYPERAUTHOR will respond:

NEW LESSON?

to which you reply:

no

You then wait a few minutes and try the lesson name and change-code again. Eventually, your lesson will be read into the workspace and you may continue.

As with SAVE, if the RESTORE job fails, there is no indication -- the file simply doesn't come back within a reasonable time! We recommend trying again if the file does not appear within two hours.

Note: SAVE and RESTORE are programs written by the C.A.T. project, and have an excellent performance record. Tape failures are the most frequent cause of errors, and can be corrected if reported immediately to the C.A.T. project. It is wise to make a safety copy roughly every five to ten SAVE runs. SAVEing and RESTOREing are expensive compared to normal student runs, but are a necessity. These programs are not maintained by the Computing Center, and all errors should be reported to the C.A.T. project as soon as detected.

All of the preceeding "lessons" deal with the TUTOR text file for a lesson. You will recall from earlier discussions that this is not the same as the condensed lesson-directory form of a lesson. Therefore, all lessons restored via "getfile" or "restore" must be read through the HYPERAUTHOR before students can run them!

OTHER INTERESTING LESSONS

MONITOR or GETDATA -- A special set of subroutines (units) and a monitoring program were devised to make data-recording for large classes easier. The units are written specially, and inserted into a lesson to be run by students. These units record information in a common block (number of right answers, wrong answers, which questions were missed, and student's comments), which may later be scanned by a proctor program.

The proctor program is not now designed to make paper reports of student progress, but this has been considered, and will be added when demanded by users.

Copies of lessons which use this capability may be obtained from the project office.

ORACLE -- In the "far out ideas" section of this publication you will find suggestions for computerized-conferencing. The ORACLE program may be copied into users' workspaces to create individual computerized-conferences. ORACLE allows many people to communicate on a kind of electronic-blackboard, either participating at the same time or over a longer time span. ORACLE has been successfully used for one nation-wide conference and a number of local conferences. It is capable of organizing several interest groups within any one conference, and maintains limited records on conference participants.

DATA -- This "lesson" may be called at any time during a run. It gives statistics on central-processor and disk usage, time logged so far and cost of the job. These figures are tabulated for the job as a whole as well as for the last lesson run. In addition, it gives a full list of current HYPERTUTOR users, and will eventually allow communications between any two.

HYPERTRANS -- The amount of work necessary to create a first TUTOR lesson is appalling! This is often the barrier which separates a good possible author from actual computer use. To eliminate (or at least reduce) this barrier, Randall Vitek, of the Dental School (on the Chicago campus) has written HYPERTRANS, a "lesson" which interacts with a lesson-author and produces finished TUTOR code from a simple interaction. HYPERTRANS allows the author to pick unit-types, create multiple-choice, fill-in, display of information and other types of units, with information s/he supplies. The finished lesson may contain answer-contingent branching, with remedial sequences of all sorts. Fairly complex structures can be created "automatically" by HYPERTRANS. HYPERTRANS is at least several times faster than manual TUTOR code-writing.

Lessons initially created by HYPERTRANS are in standard TUTOR form, and can be processed by HYPERAUTHOR at a later point.

The use of HYPERTRANS is more expensive than direct creation in the HYPERAUTHOR, but takes much less time for the author who is not willing to take extra time to learn how to program in TUTOR -- it is highly recommended that all authors try HYPERTRANS at least once.

FFAR-OUT IDEAS

Computers And Teaching, the newsletter of the Computer Aids to Teaching project, has run a number of articles on "far out" computer uses in education. Among them, the idea of computerized-conferencing as an educational tool, and the use of the computer as the data-manipulator in an electronic learning exchange. These articles from Computers And Teaching are reprinted in this chapter of the Complete Guide to HYPERTUTOR, with additons and updating to reflect more accurately the current state of the art and our thinking on these matters. As far as we are concerned, everything stated in the following pages is suggestive of things to come, and is entirely possible with existing technology.

FROM

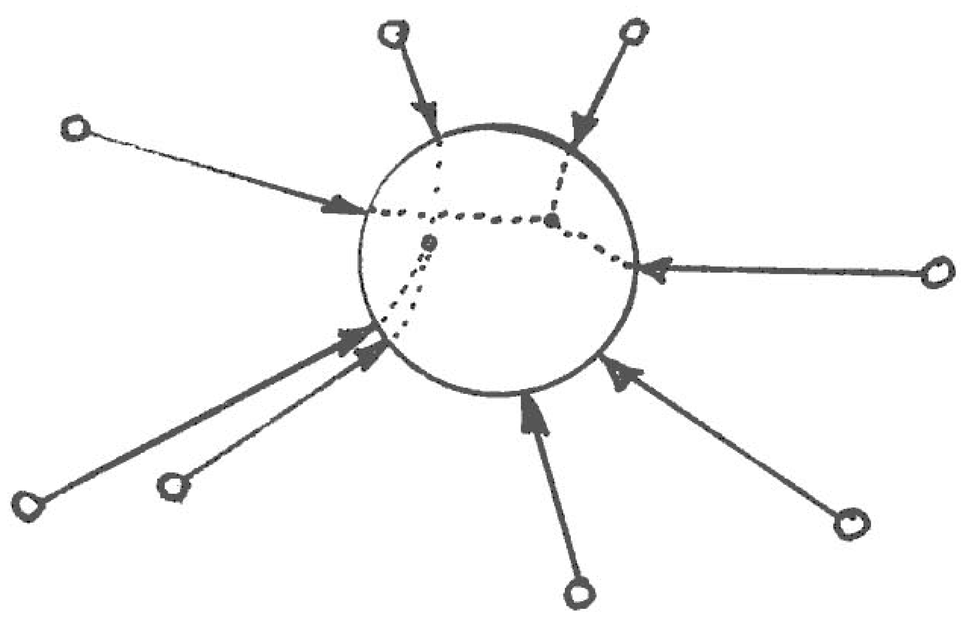
Computers And Teaching

An Interactive Newsletter

Northwestern University - Computer Aids to Teaching project

COMPUTERIZED CONFERENCING

A computerized conference is a network of people using remote terminals and a time-sharing computer as a communications device:



Interacting through a computer is very different from face-to-face interaction (as you may guess). Instead of reading the other person's face and voice, you have only the printed word to read. You must compose your thoughts carefully and then type on a computer terminal. When you have typed, the computer records your comment. If other discussion members are currently on-line, your comment is broadcast to them (unless they're typing at the time, in which case the computer waits until they're done).

People may enter or leave the computer conference at any time, and comments are preserved for others to see. A person who doesn't have anything to say can always "eavesdrop", by not saying anything. His presence is still recorded.

There are some obvious limitations on computerized conferences -- slow typing can be insurmountable. Some may find it difficult to express their ideas in writing. Many will miss the direct personal confrontation of a physical conference. But many people feel that the advantages are worth it! A computerized conference can gather together many busy people who don't have the time for travel -- it can allow democratic participation, since anonymity is possible (using pseudonyms to sign on) -- an anonymous expert must prove his worth by clarity and weighty statements, and can't rely on an overgrown reputation associated with his name. And when used on a system such as PLATO, with almost 24-hour access, it allows participation at any time of day, across the country.

The ORACLE computerized-conferencing program was used during the conference on living/learning exchanges held at Northwestern November 26, 1973, and proceedings are available in printed or microfiche form. This was its first large-scale, nation-wide test. Local groups have been using or testing ORACLE for approximately a year.

A test conference for general comments about HYPERTUTOR and C.A.I. is usually available to all users in lesson "ORACLE". This version will always contain the latest features of ORACLE (and consequently the latest bugs). Users who desire to test ORACLE's capabilities may make copies of ORACLE in their own lesson spaces.

ORACLE is made possible by the COMMON block feature of TUTOR. This makes it possible for several "programs" to communicate directly through a shared information space. A number of interesting conferencing programs exist on the PLATO-IV system, providing both the synchronous and asynchronous modes of conferencing (synchronous is like a phone call, asynchronous stores the comments for later use). We invite your participation.

A Computer-based

INFORMATION EXCHANGE

This article outlines some of the ideas and issues facing designers of computer-based learning/information exchanges. The idea of a "learning web", an information network to be used by learners and teachers in a de-schooled society, was advanced by Ivan Illich [1], in 1969. He proposed that such webs be able to link teachers with learners and learners with learners, perhaps going so far as to arrange a rendezvous. Illich recognized immediately the potential of electronic computers for storing and matching the requisite information. Silber [2], outlined a plan for a community-wide learning system which would include a computer system as the coordinating device. During 1973 a number of people have been planning or implementing such systems. Because computer based communications is such a new field, I will be stressing its advantages and disadvantages throughout this article.

BACKGROUND

In Deschooling Society, Ivan Illich argues that many of the world's problems are directly attributable to its educational structure, the school. Like other critics of the schools, Illich feels that schools (and teachers) stifle creativity, reward conformity and generally botch up the job they were commissioned to do. But, unlike other critics, Illich argues that it is the institution of "school" itself that is the source of the trouble, and that the school must be abandoned in favor of more "convivial" learning situations.

Many of Illich's critics take him to task for suggesting that the solution to the problem is the abolition of schooling. Most of them have been searching for alternatives within the system, and don't see how such a radical de-structuring as Illich suggests could be brought about without total chaos ensueing [3].

Illich *did* propose a model for learning which could replace schools, and it is unfortunate that his critics have chosen to concentrate on the unworkability of his utopian plan rather than trying to use some of the structures he outlined. The structure he proposes is the "learning web" (now more popularly called "learning exchange"). A learning-web is a network of people and information within which learners may work. In one way it is like a giant library in which people (as well as books and other resources) are catalogued. Much like a library, most learning exchanges use card catalogues to keep track of their resources.

But, Illich proposes even more. He sees the learning web as a dynamic catalog -- changing with the people who use it. It would possess the ability to schedule meetings between people who were interested in learning about or discussing specific topics. A small community could accomplish this with a bulletin board and a card catalog -- a large community might need more powerful indexing devices, such as the computer.

The Evanston, Illinois, Learning Exchange was formed several years ago by a group of students interested in testing this idea. They catalogued the abilities of each member of the group and placed this information on file. As time went by, the listing expanded and the Learning Exchange went into full-time operation. The exchange now lists thousands of people with specific skills they are willing to teach. Listings are contained in several card files. Community members call one of a number of telephones at the exchange,

are given the name and phone number of a potential match for their interest, and must then make contact on their own.

Silber's model is based upon the concepts developed above. His "Learning System" would be highly reliant upon a central computer which would hold the information currently kept in card files. This computer would be able to match teachers and learners based upon a hierarchy of subject-matter descriptions. In addition, the computer would make appointments for the teachers. Once teacher and learner had met, they would set learning-objectives to be met and would record these in the computer for future reference. At the end of the learner/teacher encounter, evaluation data would be recorded by the computer and linked to both teacher and learner characteristics. Thus, future learners would be able to determine which of a number of teachers might be best for their specific learner-characteristics (age, sex, intelligence, experience, ...). Such data would rapidly swamp a manually-operated learning exchange. (In fact, this question of evaluation of the learner/teacher encounter is being hotly debated at this moment; should there be evaluation or not?)

THE COMPUTER

The computer is really proposed only as a coordinating device for the exchange. Some people think the student should be in direct contact with the computer, and others feel that the presence of a human mediator (not a programmer, but more of a counselor) is necessary.

There are many reasons the computer is proposed, however. First, it can coordinate much more data than a human, in a given amount of search-time. The computer is also instantly available to a number of users -- a card catalog has a limited surface area, and only one or two people may be using it at a time.

A computer-terminal, which may look like an overgrown typewriter with a television set on it, could be located anywhere a telephone is available, and connected to the computer by phone whenever a learner requests information. This effectively makes the catalog of learning-resources available anywhere a computer terminal is available. The computer programs may be capable of searching through thousands of items in a second or two, thus making more complete and precise matches than a human could.

However, the computer doesn't have the ability to help a learner with a request which is imprecise. And it doesn't have the ability to associate one item of information with another on the spur of the moment, unless that association was previously noted by a human user.

TYPES OF INFORMATION

System designers often attempt to specify in advance *all* categories of information or all modes of usage for their computer systems. This is the characteristic, and perhaps the most reliable way, of producing computerized systems. However, most of these systems are inflexible, and incapable of change if the "outside world" changes. Therefore, a computer-based exchange must be organized in the most general way possible.

As a hypothetical model, I will now propose a system to be implemented within a Computer-Aided-Instruction system, the HYPERTUTOR. The system will look almost like a card-file. Each "card" is called a "record", and contains a small amount of information (perhaps 640 characters as a maximum). Each record is associated with one or more "descriptors", which are words or phrases describing the content of the record. These may also contain cross-references to other descriptors. Information on teachers or other learning-resources is entered as records, classified by existing descriptors (if possible) and may then be retrieved by users.

For example, a guitar teacher may call us and ask to be listed in the exchange. We enter his name, phone number, address and other information into a single record. We then classify this record under "guitar", "classical guitar" and perhaps "music". Any descriptor which is new to the system will be automatically added to the descriptor list (and perhaps later revised by either a system supervisor or by student users).

Several classifications of information will be designed into the system. A "teacher record" will contain information on teachers or potential teachers. Each record will be filed under proper descriptors and may be retrieved by any user. But some information in a record may be protected (for instance, address might not be released to learners, but might be available for those people who are charged with updating the information stored in the exchange). A second type of record would be the "learner-record", stored by a user who wanted to find "learning peers" to work with him on a particular subject or project. As new teacher-records are added, searches might be made to find any learners who were entered before a teacher in the appropriate category was available. A third type of record would be the "message" or "bulletin" record. These are communications which use the computer as the storage and delivery medium. A "message" is always associated with a specific destination (usually a person's name, but sometimes a place) and are designated so that they may only be retrieved by the addressee. Bulletins, on the other hand, are classified by descriptors, and may be retrieved just as any other record. They might, for example, contain a notice of a guitar concert, which could be retrieved by any interested user.

FINANCING

It might be possible to finance an exchange by donations, but experience has shown this to be unlikely. The Evanston Learning Exchange, which serves a city of about 80,000 people, would cost roughly $24,000 per year to operate manually, and that is probably a low estimate of what it would cost to run as a computer-based system. Only a small percentage is donated. Therefore, an exchange must be self-supporting to run. One potential source of income is the teachers who list themselves and charge students for lessons. They can be asked to pay a small fee (perhaps a percentage of their first lesson) which could be roughly what it would cost to take out an ad in a newspaper. (This raises the question of what a computer-based exchange offers above and beyond the classified section of the Village-Voice.)

Users searching for learning-peers may also be asked for payment, since their records require as much work and storage space on the computer as those for teachers. Users who perform searches also require computer time during the search, but require no storage space. Therefore, some charge will probably be made for searching. Will the exchange bring enough satisfaction that a person would be willing to pay about what a good paperback would cost ($1.00 to 2.00)? Which would *you* choose?

QUESTIONS

These questions and many others will be raised (and some of them answered) at a day-long conference on computer-based learning exchanges. The conference will be directed at producing a design for systems to be developed in the Chicago area, and at exposing some already conceived designs (such as the one outlined above) to constructive criticism. The FEEDBACK card at the end of this newsletter is also a viable mechanism for responding to the ideas and questions asked here. Please respond!

Jim Schuyler JAS

53

FORMAT

TYPE: { TEACHER
LEARNER
MESSAGE (PRIVATE)
BULLETIN (PUBLIC) }

CONTENT:

DESCRIPTORS:

SECURITY: { USER TYPE= LOCAT=

CROSS REFERENCES:

RELATED RECORD #=

DESTRUCTION DATE: xx/xx/xx

USES SO FAR: xxx

CONTENT: Is free-form and may be altered by originator of record.
TEACHER-record - name, (address), phone, qualifications or commer.
LEARNER-record - name, (address), phone
MESSAGE or BULLETIN - the message is the content

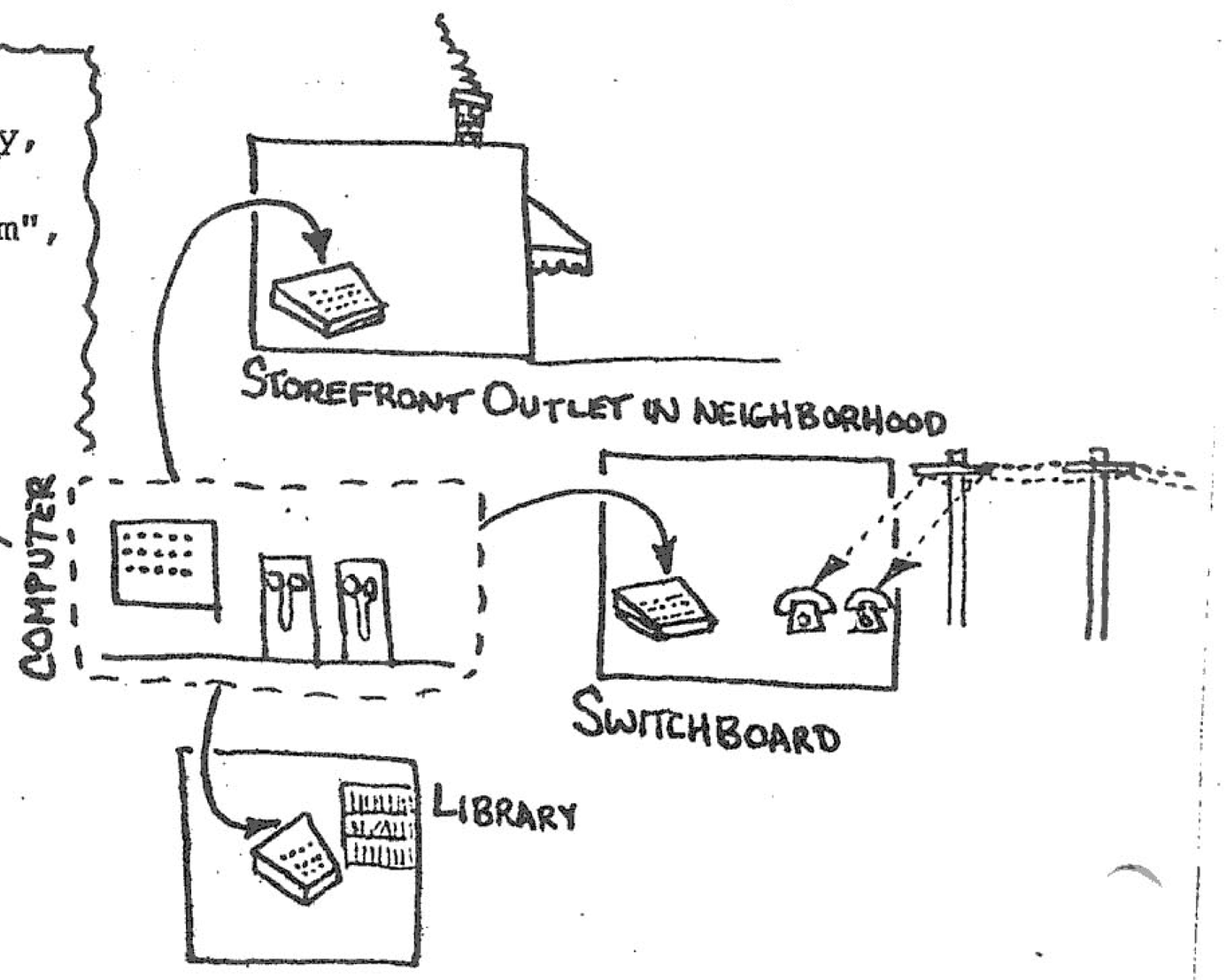
SECURITY: Record may only be retrieved by certain types or locations. For instance, a BULLETIN for San Francisco might be tagged by location.

References:

[1] Ivan Illich, Deschooling Society, Harper & Row, New York, 1971

[2] Ken Silber, "The Learning System", Audiovisual Instruction, Sept., 1972

[3] After Deschooling, What?, Alan Gartner, ed., Harper & Row, New York, 1973



COTTAGE INDUSTRY

Some people think there will be a return to the home as the economic basis of our society by the end of this century.

As our economy moves more and more toward services and farther away from manufacturing, people find themselves doing tasks they could really be doing at home, if it were acceptable. Take programming a computer -- with time-sharing, the programmer could work from a back room, and could work any hour of the night or day. Or, if one felt the desire, it would be possible to set up neighborhood work-centers, where people gathered to do their work and drink coffee. Communication would be carried out on the telephone or picturephone. "Paper" based jobs would be carried out in conjunction with computer communication systems, in which the computer stored all information. Letters would be written by typing them into the computer -- then when the recipient signed-on, the letter would be printed on his terminal. A secretary could be located in Chicago for a boss in San Francisco; the secretary would handle communications and route letters via computer to the boss, who would dictate a reply. The reply would then be played back automatically when the secretary called the boss later in the day, typed into the computer and routed to the originator.

Is this a pipe dream? Perhaps not ! There are people in telephone companies today who look forward to the day when all calls will be included in the monthly charge. It would then be advantageous to work long-distance. The telephone network would hold your calls, record callers' numbers, route your calls to another number, or "camp" on a busy line. Once the computer has been connected to this network, just imagine the possibilities !

When PLATO-IV plasma-displays get down to $700 each (roughly the cost of a color TV console), people will begin to think of buying them for their families. Learning will take place in the evening, after school and during lunch breaks. The school may have to take on more socializing tasks -- teaching kids how to deal with each other and how to settle disputes. They may begin seriously trying to direct a student's inquiry, starting neighborhood study groups oriented toward solution of local problems. And the giant communication network may be used to form larger nationwide task-forces of people, communicating via the computer.

One of the first incursions into interactive computer controlled networks is the TICCIT (say "ticket") system, by MITRE Corporation. This is centered around a cable TV system in Reston, Virginia. The cable TV will be computer controlled, and will provide information and educational materials to citizens on their own television sets, as well as regular TV fare. Signals are sent to the individual TVs in the homes of children studying lessons, displaying information much like that PLATO-IV will display on its screens. The child presses a key on his telephone to respond. Eventually keyboards will be included. Shopping can be done the same way. Items are displayed on the screen, and buttons are pushed to order. Information of community interest can be displayed, and citizens can even vote on issues.

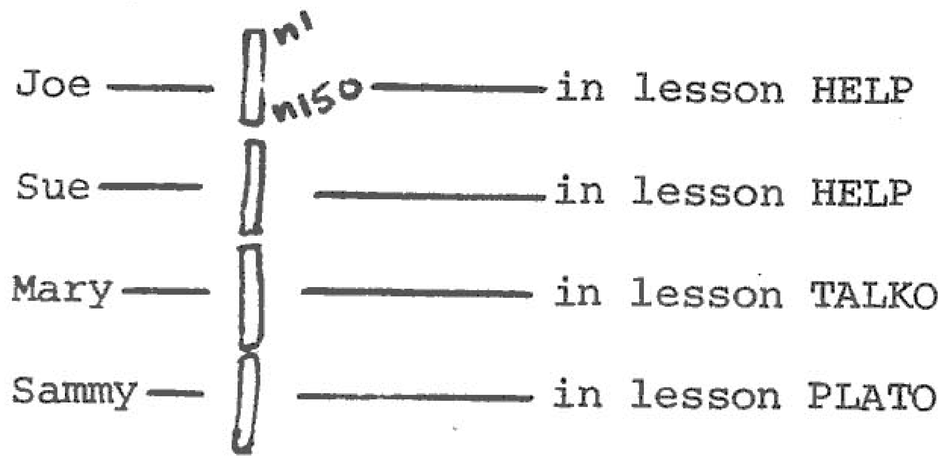
TICCIT is designed to be a local system, comprising a few hundred homes. Can you imagine what could be done on a truly large system such as PLATO-IV with thousands of homes ? It could change the structure of society.

SPECIAL FEATURES

TUTOR (when fully implemented on a computer system) has several features which distinguish it from general-purpose programming languages. One of these is its unitized structure, which governs the grouping of commands into useful entities. Another is the fact that only a small set of student variables is available for exclusive use by each student. Many features of TUTOR are occasioned by the special-purpose design of the PLATO system, but *most* can be implemented within a good time-sharing system. This chapter discusses some of the features which have been implemented for HYPERTUTOR.

COMMON STORAGE

One may think of the 150 student variables as "belonging" to each student's terminal:

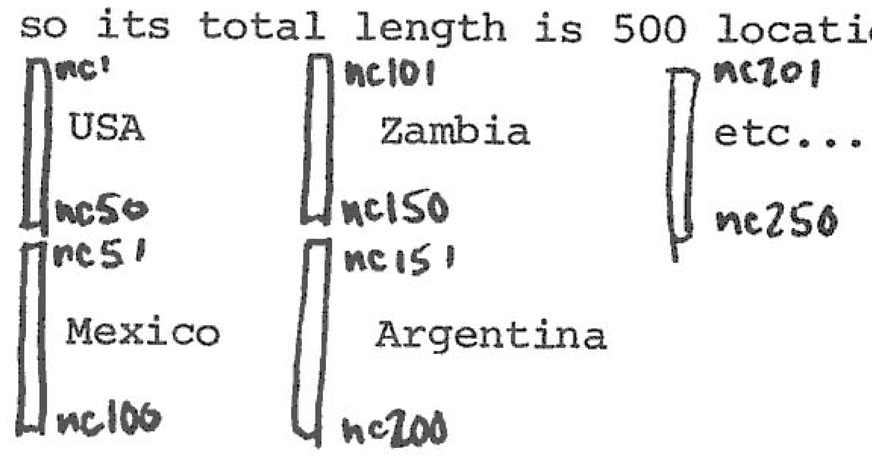


The variables are available for use by whatever lesson the student happens to be using at the time. And even though two students are using the same lesson, their variables are separate.

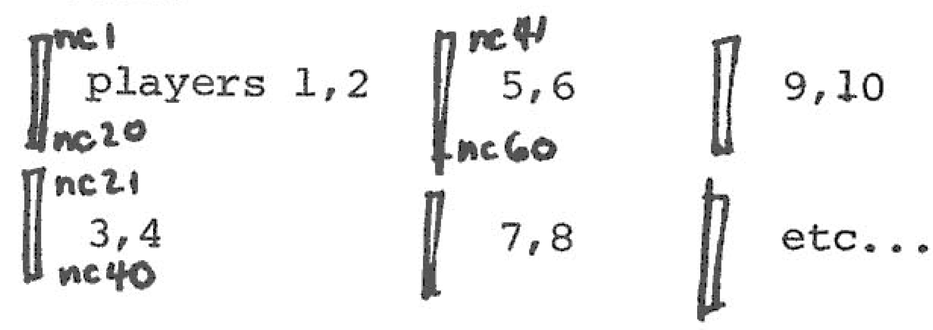
For some applications it is required that several students share a common data area, often at the same time. Some examples: two students playing a game against each other; students needing population data for a simulation; the TALKOMATIC lesson, where several students talk directly to each other; or the ORACLE, previously mentioned, in which students communicate and their comments are permanently stored.

The common storage may be of greater length than the student variables, because it is shared among all students using a particular lesson. The name and length of a common block are declared in advance by a lesson's author, and it is completely up to an author to determine how the space within a block is used. The examples which follow are only suggestions for common usage.

Assume a simple data-storage example, in which many students are to have access to population data from several countries. The author determines that 46 variables will be needed to store the data for a single country, and to allow for later insertions, allocates 50 variables to each. Data is stored in the common block for ten countries, so its total length is 500 locations.



Assume another example, in which students will be playing a game against each other. If 20 locations are needed for the data on each game, and a maximum of 10 students are to be playing, a block of 200 locations is needed.



Let's work out the actual mechanics of accessing common storage...

In the first unit of a lesson which is to use a common block, the author will place a COMMON command, including the name of the block, and its length (in variables):

```
common    pete,500
```

[Note: On PLATO-IV one includes the name of the lesson in which the common is to be stored, and the block name. Because common storage is kept separate from lessons in HYPERTUTOR, the names may be independent of lesson names.]

If no previous references have been made to the common block, it will be created when the lesson is first run, and will be filled with zeros.

Common storage is accessed much like student variables, using a format type, "N" or "V", the letter "C" (to indicate common) and a number telling which location is to be referenced in common. Indirect accessing is, of course, permitted for common too. Common variables may be used anywhere student variables would be used. Here are some samples:

```
calc      ncl=0
calc      nc(ncl)=nc(ncl)+1
show      nc(n44)
```

Returning to the two examples on the last page, imagine that the student of population is to pick a country, and the lesson then plots some information on the screen. Information on the student's choice must be kept in student variables, and the most common technique is to store a "pointer" which indicates which part of common the student is working with.

```
unit      xxx
at        503
write     Which country would you
          like to study?
           1. U.S.A.
           2. Mexico
           3. Zambia
           4. etc...
arrow     3003
answer    (1,USA,U.S.A.)
calc      nl=1
answer    (2,Mexico)
calc      nl=51
answer    (3,Zambia)
calc      nl=101
answer    (4,Argentina)
calc      nl=151
...etc...
```

The value of "nl" will depend upon the student's selection. Because each of the country data areas in the common block is 50 locations long, nl will correspond to the variable number of the first location in the country chosen by the student.

If the first item in each country's data area were its population, it might be shown as follows:

```
unit      pop
at        503
write     The population of your
          chosen country in 1973
          was
at        707
show      nc(nl),10
```

Note that indirect addressing is used to obtain the proper data for whatever country is being studied. If the student picks the USA, nc(1) will be shown, and if s/he picks Argentina, nc(151) is shown.

A similar strategy can be used to assign storage in the game-playing example. In most games, however, an additional tab of players' names is included somewhere in common storage. This makes it possible to display all names so a player can make a choice of partner.

Common blocks are used for data storage by the MONITOR or GETDATA lessons, and each student in those cases is allotted a fifty-word block. Allocation is done at the time of data recording, however, and ncl always is set to tell which location in common is the next one available for data storage.

HYPERTUTOR keeps common blocks in disk storage. Because of this, it is possible that common blocks may be "idled" just as lesson text is idled (see the section on special lessons). To keep this from happening, authors should ensure that their lessons which use common storage are run at least once every few days.

RANDOM NUMBERS

TUTOR provides three distinct ways of generating "random" numbers for use in lessons. It can a) produce a number between 0.0 and 1.0, b) produce a number between 1 and some chosen positive integer, or c) create a permutation of the numbers between 1 and some chosen positive integer which can later be examined.

Numbers in the range 0.0 to 1.0 --
A "uniform" random number generator is invoked by the command:

```
randu     v1
```

and its result (a number between 0.0 and 1.0) is stored in the variable mentioned in the tag field.

Positive integers -- The uniform generator can be used to produce integers:

```
randu     n1,10
```

The variable mentioned first receives the resultant number, which is between 1 and the value of the second expression in the tag (in this case "10").

Permutations -- The SETPERM command sets up a permutation of up to 120 numbers, which may be sampled at a later time. The tag of SETPERM indicates how many numbers are to be created:

```
setperm   n55
```

(In the example, the value of n55 is taken, and a corresponding number of terms is then permuted.) Assume that n55 equals 5. The numbers 1 thru 5 are essentially put into a "bucket" and scrambled. When a RANDP ("P" for permutation) command is executed, a single number is removed from the bucket, and returned in the variable which appears in the tag:

```
randp     n56
```

The results might be 1, 5, 2, 4, 3, or any combination of these numbers. If extra RANDP commands are executed, the value "0" will be returned.

If desired, the permutation may be recreated before all numbers have been sampled, or a new permutation with a different number of elements may be created.

Here is an example in which RANDU is used to determine which of three possible questions is to be asked of a student:

```
unit      xxx
at        503
write     Answer the following question:
randu     n100,3
inhibit   erase
jump      n100,x,x,xxx1,xxx2,xxx3
unit      xxx1
at        603
write     Question 1...
...etc...
next      yyy
unit      xxx2
write     Question 2...
...etc...
next      yyy
unit      xxx3
write     Question 3...
...etc...
next      yyy
unit      yyy
at        503
write     Next question:
...etc...
```

The permutation generator can be used to randomly order a set of questions or parts of a lesson, as shown by the example on the next page.

```
unit      xxx
at        503
write     You will now see the names
          of four colors:
setperm   10
next      select
unit      select
randp     n99
jump      n99,x,done,1,2,3,4
unit      1
at        1003
write     blue
next      select
unit      2
at        1003
write     orange
next      select
unit      3
at        1003
write     purple
next      select
unit      4
at        1003
write     pink
next      select
unit      done
at        1103
write     Wasn't that grand?
          Press NEXT to try again.
next      xxx
```

This series of units can be executed again and again, and many different color patterns will be created.

The random number generator is "seeded" with the time of day at the beginning of each lesson, and therefore a lesson will almost always create different series of numbers for different students.

JUMPOUT

For many reasons, it is often necessary to transfer control from one lesson to another. The JUMPOUT command allows this to be done. The tag of a JUMPOUT may contain a lesson name, or a lesson and unit name -- control is transferred directly to the lesson (or to the unit within the lesson) named. These lesson and unit names always appear in their quoted forms, for clarity:

```
jumpout   "trials","sincere"
```

This example jumps to unit "sincere" of lesson "trials".

JUMPOUT also may appear in a conditional form, with quoted lesson names in the list.

A lesson which has been reached via a JUMPOUT has certain information available to it which permits a return to the original lesson. First, the FROM command may be executed to determine which of a number of possible lessons was the student's original lesson. Its form is described in the command summary. This information can then be used to initiate a JUMPOUT back to the original lesson. Or, a:

```
jumpout   return,return
```

can be used to return control to the unit following the JUMPOUT in the original lesson. This provides no information to the secondary lesson about where the original JUMPOUT came from (which is exactly what the designers of PLATO intended).

VOCABULARIES

Answer-checking can become tedious when there are many synonyms for each possible answer -- in TUTOR, they have to be listed explicitly in each ANSWER or WRONG command, enclosed in parentheses:

```
answer    (yes,sure,certainly)
```

However, using the CONCEPT command, it is possible to specify vocabularies of synonymous words and words to be ignored. These lists are then used in all subsequent CONCEPT commands, obviating the need for repeating long lists of synonyms.

Specification is done in a VOCAB or VOCABS command:

```
vocabs    <a,an,the,of,some,this>
          (blue,azure)(red,scarlet)
          (camel,dromedary)
```

Words within parentheses are synonyms. Words within angled brackets may be ignored during judging. Other words not within either parentheses or the angled brackets are significant words which may be used in CONCEPT commands and *all words to be used in any CONCEPT command must appear in a VOCAB or VOCABS.*

VOCAB from last page used

```
unit      xxx
at        503
write     What is the rarest type
          of African animal?
arrow     703
concept   red dromedary
write     How did you know that?
concept   red monkey
write     Wrong. Guess again!
judge     no
no
write     This really isn't fair,
          is it?
```

Answers such as:

the scarlet dromedary
a red camel
some red camels

...will all be accepted. In the first two cases, acceptance is based upon the synonomy of red and scarlet, and camel and dromedary. In the third case "camels" is accepted as a misspelling of "camel" (actually detected as a suffix by HYPERTUTOR).

Any of the words from a synonym group (or "ring" as it is called in HYPERTUTOR) may be used in the CONCEPT command -- it will match any other word in the group which the student might use in an answer.

The PLATO system will judge "no" any answer which partially matches a CONCEPT command but includes words not found in the VOCAB or VOCABS. Also, a CONCEPT command is not allowed to contain any words not listed in the VOCAB or VOCABS. This is not true of HYPERTUTOR.

NOTES ON PROGRAMMING TECHNIQUE

There are some additional hints to be given here which actually don't involve special TUTOR capabilities. One of these which novice authors often encounter is the interaction in a multi-ARROW unit.

```
unit      xxx
write     First question...
arrow     503
answer    ...
...
wrong
...
no
...
endarrow
write     Second question...
arrow     1503
answer    ...
...
wrong
...
no
...
```

The critical point in this example is the appearance of ENDARROW at the end of the answer-checking commands for the first arrow. If the first ARROW is answered correctly, ENDARROW permits regular processing to begin again, thus allowing the writing of further information or questions. If the first ARROW is not answered correctly, ENDARROW indicates the scope of that ARROW has been reached and sends the student back to repeat. *Without* the ENDARROW, the WRITE which follows would be contingent upon the NO command, rather than appearing when the student gets to the second ARROW.

COST

Some rough estimates on the cost of operating HYPERTUTOR can be made on the basis of six-months' experience with intensive student use. First, the cost of equipment:

Terminals:

Hazeltine-2000	about $2,200
A.D.D.S. 580	about $1,800
Texas Instr. 725	abt $2,790
Texas Instr. 733	abt $2,000
Plato-iv	about $5,500
Plato-iv (from Magnavox)	about$10,000

Modems:

Acoustical coupler	$ 300
Plato-iv modems	$ 600

Telephones:

Regular on-campus phone (local only) per mo.	$ 10

Renting terminals:

Hazeltine-2000 per mo.	$ 88
Teletypes per mo.	$ 50
Texas Instr. 725	$ 140

A.D.D.D.S. can't be rented

Plato-iv can't be rented (except Control Data may be renting them soon)

Computer time -- This is absolutely the most variable factor in the cost equation! Computer time will vary from lesson to lesson, primarily as a result of varying lesson strategies. Inquiry mode lessons will be expensive, with lessons where the student has to do a lot of thinking between questions the least expensive. Lessons which perform many calculations will also be expensive. Computer time on the PLATO system is measured in terms of the number of milliseconds per real-time second the CPU is used by a lesson. On HYPERTUTOR systems, PPU time is also used, and the CPU time doesn't provide an adequate measure of cost. Many systems charge high rates for PPU time -- PLATO effectively ignores it.

At Northwestern, computer time is charged for by the CPU hour, plus 20% of PPU time, all multiplied by a factor calculated from the amount of central memory the job uses. A HYPERTUTOR job uses little central memory, and is charged correspondingly less than large-core systems such as PLANIT. Figures on the following pages have been computed for Northwestern's charging system, and as time goes by we will accumulate more figures from other systems, which will be added to this document.

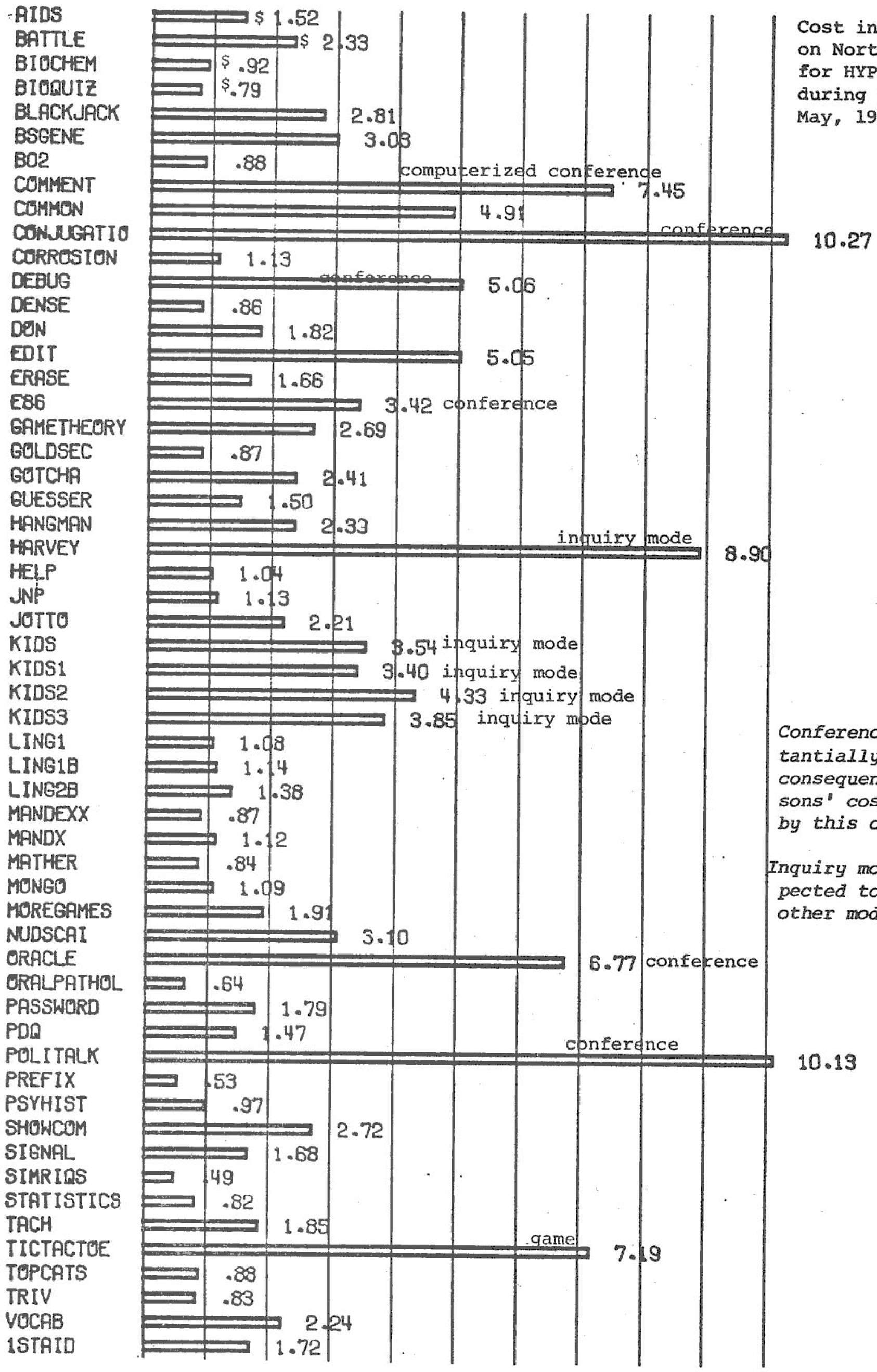
Lessons chosen for these charts were all run in the past six months, while HYPERTUTOR was recording efficiency figures for all student runs. During that time, thousands of student jobs were run, each referencing several lessons.from the HYPERTEXT. Because it is fairly expensive to initialize a lesson, if a student just starts a lesson and almost immediately terminates it, we excluded the figures from the charts -- *we wanted the charts to represent actual student use assigned as part of classwork,* and this normally does not allow browsing. Therefore, all lessons which ran less than one full minute were eliminated from our figures.

Author using the HYPERAUTHOR lesson account for a large amount of C.A.I. computer time. Those figures were also separated from the charts -- the HYPERAUTHOR, during the six month test period, was used 4235 times, a total of 44,663 minutes, and its average cost per hour was $5.78, much more than student costs. (Student costs averaged $1.70 per hour.) With authoring costs and casual use included, HYPERTUTOR cost an average of $4.24 per hour.-- this figure is so high compared to the student cost because of the predominance of authors during the developmental phase, during the past year.

(61)

HYPERTUTOR average computer time costs (not including phone, connect charges, computer terminal, etc.) for *selected lessons*, from approximately October, 1973 through April, 1974: Student use only ---

name	uses	minutes	cost/hour	cpu msec/sec	average minutes used
bioquiz	23	558	.79	.5	24.3
blackjack	236	2133	2.81	2.8	9.0
bsgene	23	204	3.03	5.8	8.9
conjugation(confer-	44	302	10.27	5.3	6.9
dense ence)	96	507	.86	.7	5.3
e86 (conference)	44	286	3.42	3.5	6.5
gametheory (games)	88	1377	2.69	2.3	15.6
goldsec	22	157	.87	.9	7.1
hangman (game)	85	668	2.39	3.5	7.9
jotto (game)	197	1990	2.21	2.8	10.1
kidsl (inquiry)	26	186	3.40	4.6	7.2
mather	21	236	.84	.6	11.2
moregames (games)	45	290	1.91	2.1	6.4
password (game)	130	1258	1.79	2.2	9.7
prefix	25	237	.53	.4	9.5
simriqs	36	375	.49	.4	10.4
statistics	49	374	.82	.6	7.6
topcats (game)	63	727	.88	.7	11.5
vocab	44	429	2.24	2.7	9.8
*b02 quiz 1	56	256	1.47	.7	10.3
*b02 quiz 2	249	2577	1.07	.7	11.0
*b02 quiz 3	164	1795	1.04	.6	10.9
*b02 quiz 4	256	2192	1.12	.6	8.6
*b02 quiz 5	226	2723	1.12	.7	12.0
*b02 quiz 6	225	2481	1.06	.6	11.0
*b02 quiz 7	254	2476	.99	.6	9.7
*b02 quiz 8	284	3126	.94	.5	11.0
*b02 quiz 9	172	1862	1.04	.6	10.8
*goldsec	45	577	.79	.8	12.8
*latin	21	100	.87	.5	4.8
*ldchild	78	469	.92	.5	6.0
*primer	34	288	.78	.5	8.5
*rationals	39	484	.53	.3	12.4
*sbg	47	1906	2.25	1.5	40.6
*synth	44	281	.94	.6	6.4
*firstaid	43	550	.91	.7	12.8

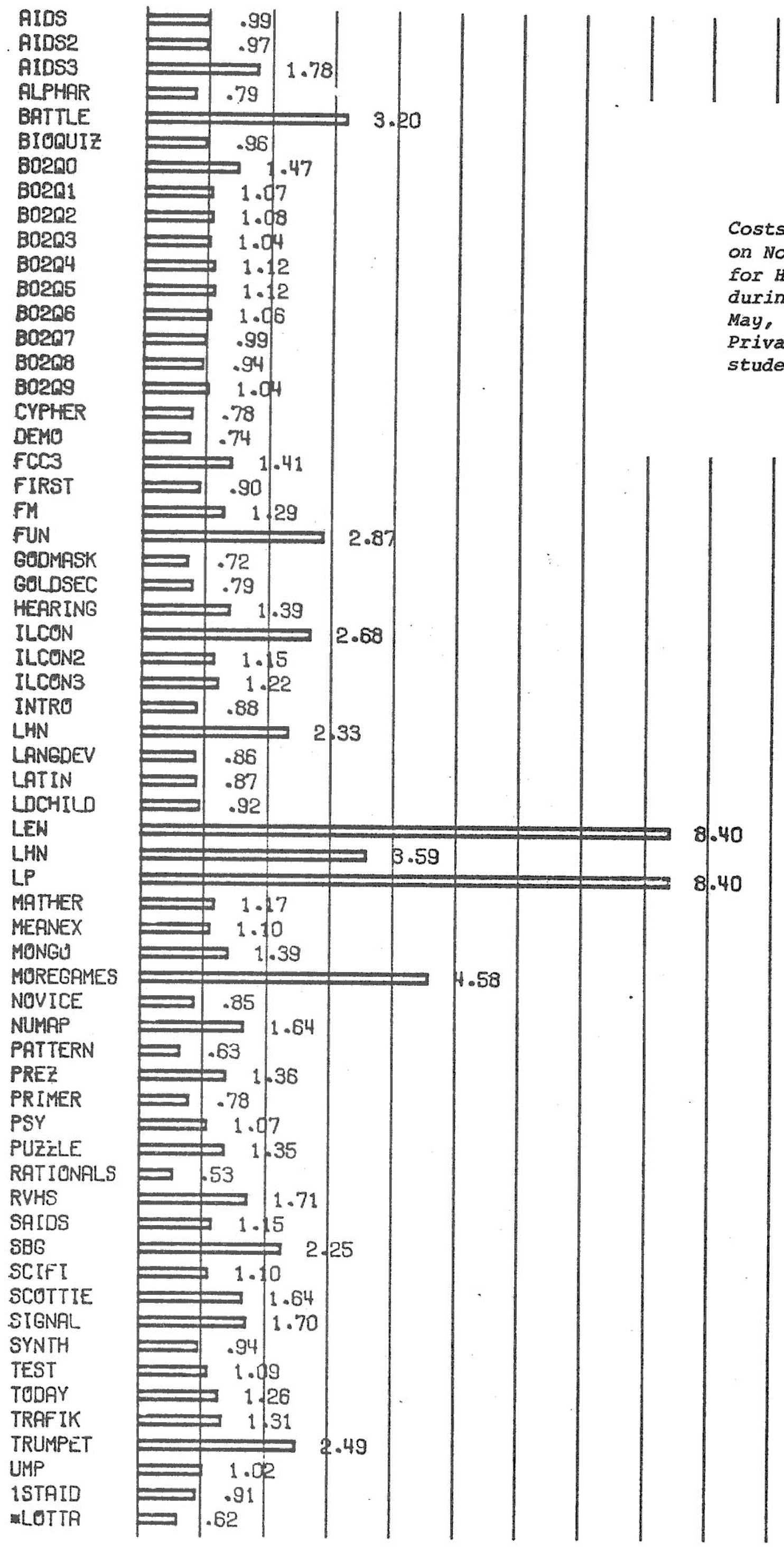


Cost in dollars per hour, on Northwestern's CDC 6400, for HYPERTUTOR lessons, during October, 1973 thru May, 1974 test period.

Conferencing costs were substantially reduced in May, and consequently conferencing lessons' costs are misrepresented by this chart.

Inquiry mode lessons are expected to cost more than other modes of interaction.

63



Costs in dollars per hour, on Northwestern's CDC 6400, for HYPERTUTOR lessons, during October, 1973 thru May, 1974 test period. Private lessons only -- student use.

SYSTEM STRUCTURE

This section deals with some of the special system-related portions of the HYPERTUTOR system. It will be of interest to systems programmers, and to those considering the implementation of HYPERTUTOR on their own CYBER 70 series computers.

HYPERTUTOR is an interpreter which runs on CYBER-70 computers, accepting the TUTOR-IV (PLATO-IV) language. It runs a range of computer terminals, from the full PLATO-IV terminal (through a special device) down to CRTs and printing terminals, where these would be appropriate. The range of graphics depends upon the type of terminal used, and the range of interaction upon the operating system. It is called HYPERTUTOR not because it is an improvement upon TUTOR itself, but because it incorporates lessons into a scheme called a Hypertext (invented by Theodor H. Nelson).

Rationale -- There are many CYBER-70 installations which would like to make use of courseware written for PLATO-IV for various reasons. The high cost of PLATO-IV terminals makes a system attractive which can also run existing terminals, but whose language is compatible with the full PLATO system. And budget considerations for some universities may dictate on-campus expenditures for computer time, at existing installations, rather than connection to PLATO-IV in Urbana. Users who write lessons locally may still want upward compatibility with the PLATO system, to increase the marketability of their software. Last, but not the least, users will want these capabilities at the lowest cost. The HYPERTUTOR has been designed with all of these considerations in mind.

STRUCTURE

HYPERTUTOR uses a "condensor" program to prescan the TUTOR input and detect syntax errors. This creates a lesson-file which is stored as a Scope permanentfile, and information which is stored in a *lesson directory* for later use. The HYPERTUTOR interpreter then works with this precondensed version when a student is on-line. Lessons are stored on the disk, and are paged into central memory as needed. The system can use Extended Core Storage (ECS) if it is available for direct access from on-line jobs. It also takes advantage of multiple-access permanentfiles if they are present, to implement the *common block* structure. HYPERTUTOR runs one job per student on the computer, and does not yet take advantage of multiple-user-job (MUJ) capabilities of the operating system.

PLATO COMPATIBILITY

The important question of compatibility with PLATO-IV is being explored at the present time. PLATO programs have in some cases been directly transferred to HYPERTUTOR via magnetic tape, with no changes in TUTOR text required. In many cases, PLATO-IV capabilities (such as individual keystroke processing) could not be implemented on the Scope operating system, and these lessons were not transferrable. Because of the limited graphics capabilities of most CRT terminals, most lessons are useful only when run on the PLATO-IV terminal, and cannot therefore be transferred to all CYBER systems.

Because of the importance of these special PLATO-IV features in the operation of many lessons, HYPERTUTOR is not in any sense "guaranteed" to accept PLATO-IV programs and run them on limited CYBER systems. In the other direction, however, we attempt to guarantee that lessons written to run on the HYPERTUTOR will run exactly as written on PLATO-IV systems.

Because of the changing and experimental nature of the PLATO-IV system, HYPERTUTOR often is a few months behind the PLATO system in implementation of new commands or features. In some cases, new features are not implemented on HYPERTUTOR.

COMMON STORAGE

The creation of TUTOR common storage depends upon some form of storage which may be accessed by several jobs at the same time. Systems which provide permanentfiles which may be accessed (for both reading and writing) by more than one job at a time, can successfully utilize common storage.

Common is divided into 64-word pages, which are swapped into and out of two buffers within a HYPERTUTOR job. Manipulations which take place within a single page do not require disk accesses. The presence of two pages at a time ensures smooth transfers of blocks of storage by the BLOCK transfer command. Common storage may also be placed in commonly-accessible ECS blocks, if these can be made available on the system in question.

HYPERAUTHOR

The HYPERAUTHOR requires direct access to the system Editor program. This is provided at Northwestern, and at the University of Texas (Austin), the second HYPERTUTOR installation. Files of lesson text are therefore created directly in a familiar Editor, and are stored in whatever locally available form the user requires.

ACCESS TO OTHER PROGRAMS

HYPERTUTOR, as implemented at Northwestern, allows users to switch execution to other main programs (such as BASIC), and then to return to HYPERTUTOR at an appropriate time. This is done by saving an "image" of central memory, then calling the required main program, finally regaining control (thru a mechanism called "reprieve") and restoring the central memory of HYPERTUTOR so it can continue. If this capability is available, users may freely switch to other programs in other languages at any HYPERTUTOR installation.

LESSON DIRECTORY

The lesson directory is implemented as a direct-access ("random") file on the Scope system. An index, containing the names of all lessons, is kept as the last record in that file. Along with each name is an indication of the relative position of its Hyper-commands, which are kept within the information file. This permanentfile is named HYPERINFO, and is created automatically by the condensor as the first TUTOR lesson on a system is condensed. Private lesson names within this directory are never displayed in the browsing mode, but information is kept nevertheless.

CONDENSING

The condensation process takes place for each and every lesson entered into the lesson directory. A program called READIN, accesses a disk-resident copy of the TUTOR text for a lesson, scanning each command for obvious syntax errors. Error messages are issued on-line, and a printout is created for every condensation. The printout, on the OUTPUT file, may later be sent to a printer on the computer (this is done by a special job, PRINT, or by HYPERAUTHOR, at N.U.). READIN does not require line numbers, but will use them to create a reference map for all units in a lesson -- the line numbers must appear in columns 73-80 of each line.

The condenser, READIN, creates a second copy of the lesson, separated into the actual units -- each unit is a separate logical record in this copy. Command names have been separated, and each is followed by a *pointer* to the next command in the unit, so that skipping from one command to the next is extremely fast. CALCs and expressions are not precompiled (as on PLATO-IV). An index is placed at the end of the lesson file which contains all unit names, plus some information about any VOCABS and TERM commands found within the lesson. This final file is catalogued by HYPERTUTOR, as a regular permanentfile, with special control passwords so that only HYPERTUTOR can purge it. In addition, the PASSWORD command, if present in the Hyper-commands for a lesson, is used as a special password on the lesson, giving added protection.

As an author updates a lesson, s/he will condense additional versions of it into the lesson directory. For each new version, hyperinformation is updated (changed), and the lesson file is recreated. If the lesson file is being used by one or more students, a new "cycle" is created by the operating system, and students who then try to use the lesson will get the later version (even though students using the lesson while the author is working may continue working with their old copies).

CHECKUP

Because lessons and lesson text will be "idled" to magnetic tapes after a period of inactivity, a special "lesson" called CHECKUP is run by the HYPERTUTOR supervisor every two days. This lesson uses the information kept in the lesson directory to activate each and every lesson (including the private ones), so that it will not be idled. Because this procedure depends upon human intervention, it can be forgotten, and occasionally lessons will be idled. In fact, at the end of an academic quarter, the CHECKUP is often omitted on purpose, to remove those lessons not in active use.

DAILY HYPERINFO

Because the HYPERINFO file is being used by READIN (the condensor) as a permanent repository of information about lessons, it must be able to gain complete access to the file at any time during the day. The Scope system will not allow this if several other users (students) are also accessing the file. Therefore, each day, at the start of operations, a separate copy of HYPERINFO is made, called DAILYINFO, which is then used by all students during the remainder of the day.

Because of this, new lessons inserted into the Hypertext during the day will not be available in the browsing list until the next day, when DAILYINFO is updated. However, because the lesson information is available immediately on a permanentfile, students who know the name of a lesson may run it as soon as it is condensed into the lesson directory. This also makes it possible for authors to immediately test their lessons.

"PLATO" VERSION

A separate version of the HYPERTUTOR interpreter is used to drive the PLATO-IV terminals. This version is called automatically, when you tell the computer you are running a PLATO-IV terminal, at the start of a C.A.I. job. The special version will also run regular CRT and printing terminals, if necessary, but it is larger, and response time may be slightly worse than for the regular HYPERTUTOR.

Normally, both versions of HYPERTUTOR will contain new modifications, and it will be safe for students to assume that either will work at any time.

THE NUBB

The HYPERTUTOR communicates with the PLATO-IV terminals through a device called a NUBB (Northwestern University Black Box), which changes ASCII codes into PLATO codes, and vice versa. As the student presses keys, impulses are sent to the central computer, where they pass through the NUBB, are converted to ASCII, and go into a PDP 8 computer which directly drives the PLATO-IV terminals. This PDP 8 communicates with the CDC 6400, where HYPERTUTOR is running. When HYPERTUTOR creates information for the terminal, it checks the terminal type indicator stored internally; if it indicates a PLATO terminal, codes are converted into PLATO codes, and are sent to the PDP-8, which converts them first into ASCII and sends them to the NUBB; the NUBB then converts to PLATO terminal codes and sends the information to the final destination, the terminal. The student sees things exactly as if sent by PLATO-IV itself.

MERLIN'S DATA

The response-checking part of HYPERTUTOR is called MERLIN. It performs all analysis of student responses, except for numeric answer-checking. MERLIN records efficiency data for each lesson HYPERTUTOR runs, and at the end of every lesson, MERLIN sends this information to a special permanentfile, MERLINDATA. This file is extended by one record for every lesson run by HYPERTUTOR. [Note: An installation may turn off this data-recording option, if desired.]

The data can later be analyzed and charts like the ones in the last chapter, indicating hourly costs, CPU usage, etc. can easily be created.

The analysis program is not available to normal users of HYPERTUTOR, but can provide statistics on individual lessons or on aggregated lessons. It is used to test new HYPERTUTOR "improvements".

HYPERCOMMUNICATIONS FILE

While HYPERTUTOR is active, a special communications file is shared by all HYPERTUTOR jobs. This file gives information on all currently active C.A.I. jobs, including the name of the user, the job-name, the lesson being used and the time that lesson was initiated. The file is updated each time a user changes lessons. Part of the content of the communications file is printed by the DATA lesson. At some future date it will be used as a direct communications link between users, allowing any user to directly contact any other user.

FILE READING AND WRITING

HYPERTUTOR (and PLATO-IV) use disk files sparingly, because they involve relatively slow operations, which would slow the pace of the C.A.I. system. In some cases, disk accesses cannot be avoided, however, as in the creation of lesson text, or in the access to common blocks. Obviously, therefore, TUTOR has a way of accessing the disk. This mechanism allows direct access to any record of a file, and is used by some system-programs which require access to student records, or to storage spaces for lesson text. Information can be provided on this capability for those who need it.

Because HYPERTUTOR is maintained by the Computer Aids to Teaching project, all information of a technical nature should be requested from C.A.T. at the address given on the front cover.

COMMAND SUMMARY

Each HYPERTUTOR command is profiled below. This section of the GUIDE is for reference, after you know how to write simple programs -- it will not teach you how to write lessons. As new TYTOR-IV commands are added to the PLATO-IV system, we examine them and attempt to add those we can use. We try to maintain upward-compatibility with PLATO at all times.

HYPERTUTOR commands begin in column 1 -- the tag field begins in column 11.

ABORT

SAMPLE: ABORT COMMON
TAG: The word "common" or "record".
PERFORMED: This command, as well as most of the commands which follow, will be performed when encountered in the normal execution of a lesson. It is called a "regular" command. Non-regular commands will be noted.
*** EFFECT: *ABORT COMMON will not work on Hypertutor (it is noted only for compatibility with PLATO-IV)*. ABORT COMMON allows the lesson to continue but the common-block currently in use is not returned to the disk at the end of lesson execution, as it would normally be.
ABORT RECORD aborts student record-keeping.

ADD1

SAMPLE: ADD1 V6
TAG: A HYPERTUTOR variable of any sort (not an expression!).
EFFECT: This command adds 1 ("V" or "N" type) to the named variable.

ANSV

SAMPLE: ANSV N5,9
ANSV SQRT(V8*3),3%
TAG: First type -- a HYPERTUTOR expression, comma, second expression. Indicates an absolute tolerance (see below).
Second type -- a HYPERTUTOR expression, comma, second expression followed by percent sign. Percentage tolerance.
EFFECT: First type -- the student's answer is checked against the value of the first expression. The second expression indicates how far the student's answer may lie on either side of the indicated correct answer and still be judged "ok". In the sample above, the answer must be within 9 of the value of N5.
Second type -- the student's answer must be within the indicated percentage range of the indicated correct answer. In the sample, the answer will be accepted if it is within 3% of the value of SQRT(V8*3).
NOTE: The WRONGV command may be used to judge responses "no".

ANSVA

SAMPLE ANSVA N40,6
TAG: A HYPERTUTOR variable (not an expression), comma, an expression.
(more on the next page)

(ANSVA continued)

EFFECT: Takes characters stored in the variable and checks them against the student's answer, one by one. The expression indicates how many characters must match (from left to right) before the answer is judged "ok". If the expression is not given, a length of 10 is assumed. If the length given is greater than 10, the matching process will continue into successive HYPER-TUTOR variables, 10 characters per variable.

ANSWER

SAMPLE: ANSWER (LOVE,AFFECTION,RESPECT)

TAG: A list of words or phrases which constitute correct answers to the last question asked. The parentheses are used to enclose a set of synonymous answers, which are then listed and separated by commas -- any of the answers thus listed will be accepted as correct. Because parentheses are used to enclose synonym lists, they cannot be required as part of the correct answer.

PERFORMED: Only following an ARROW command. The correct structure is illustrated earlier in this booklet. A number of ANSWER and WRONG (or other response-judging) commands usually follow a single ARROW, and they may be intermixed. Response-judging commands may be in JOINed units.

EFFECT: Three levels of response-checking are performed by ANSWER.

Level one -- the answer=judging routine first searches for an *exact match* between answer-judging commands and the student's answer. If one is found, processing proceeds to the command immediately following the matched answer command. The student's response is judged "ok", and he may proceed to the next question or unit.

*** Level two -- spelling mistakes are allowed. The student is not informed about such errors. *(On PLATO-IV spelling mistakes are judged "no" and the student must repeat.)* If the author has specified a VOCAB command, it will be used at this level.

*** Level three -- the answer-judging routine searches through the student's response for the occurrence of the word(s) in the ANSWER command. If they are found, the response is judged "ok". *(This is not done on PLATO-IV.)* This is called a "keyword" check, because the appearance of the word specified (no matter where in the student's response) triggers

NOTE: ANSWER *must* have a non-blank tag. Older TUTOR users will recall that a "universal" ANSWER used to be available -- it is no longer: instead, the OK command must now be used.

ARROW

SAMPLE: ARROW 1205

TAG: An expression giving a *coarse* screen position.

EFFECT: The appearance of an ARROW command first terminates any answer-judging for previous ARROWs in the same unit. Previous ARROWs must be judged "ok" before the next arrow will be processed.

An arrow is plotted on the screen at the indicated position. Coarse screen positioning is explained under "AT" (below).

(more on the next page)

(ARROW continued)

After the arrow is plotted, processing continues until an answer-judging command is reached. (Regular commands are therefore processed before the student is required to answer.) When the first judging command is reached, the computer pauses and waits for a complete answer, signalled by the NEXT key (or its equivalent on the terminal being used. Then, processing returns to the ARROW command and only the answer-judging commands are processed. A number of commands such as OPEN, CLOSE, STORE and STOREA will routinely be located between an ARROW command and its first associated answer-judging command.

***

HYPERTUTOR does not limit the number of ARROW commands which may appear in one unit; PLATO does, and this limit is currently 12. ARROWs *must not* be placed in JOINed units.

[*] (asterisk)

SAMPLE: * this is a comment

TAG: Anything.

EFFECT: This command indicates that you wish a line to be inserted as a comment. The computer will not process that line, but will print it with your lesson's listing.

[AT]

SAMPLE: AT 1205
AT 0,N5

TAG: Coarse or fine screen position -- this may be any HYPERTUTOR expression.

EFFECT: The current screen position is changed to the one specified. The *coarse* coordinates may be used on CRT terminals other than the PLATO-IV terminal, or on the PLATO-IV terminal. The *fine* coordinates may only be used on PLATO-IV terminals. COARSE coordinates -- the "hundreds" part of the expression indicates the line number, the "tens" and "units" combined indicate the character position on that line. The first example above indicates line 12, character position 5. The AT command also sets a left-margin indicator, and subsequent *paragraphed* WRITE commands will be positioned at that margin. FINE coordinates -- the first expression indicates the "X" (horizontal) position. The left edge of the plasma panel is "0", and the right edge is "511". The second expression indicates the "Y" (vertical) position, with "0" at the bottom of the screen and "511" at the top. Characters are 16 dots vertically and 8 dots horizontally, compared to the fine coordinate system.

[AUTHOR]

SAMPLE: AUTHOR Jim Schuyler (492-5367)

TAG: Author's name and telephone number.

*** EFFECT: *This command is not available on PLATO-IV.* It indicates the name of the lesson's author, and is permanently entered in the HYPERTEXT organizational scheme for use by students. It may be updated by simply changing the command in the lesson. AUTHOR and other HYPERCOMMANDS *must* appear before the first UNIT of a lesson.

AUDIO

This command is ignored by HYPERTUTOR. Audio units are not yet available for the PLATO-IV terminal.

AXES

SAMPLE: AXES 100,100
AXES -100,-100,100,100

TAG: First type -- two expressions giving upper limits on X and Y axes.
Second type -- four expressions giving lower X and Y and upper X and Y limits.

EFFECT: A set of axes will be drawn on the screen, passing through the current ORIGIN (see ORIGIN command) and extending to the limits set by AXES. The length of these will be affected by the current scale. These only work on the PLATO-IV terminal.

BACK

SAMPLE: BACK JONES

TAG: A unit name, or a conditional list of unit names.

EFFECT: If a BACK command exists in a unit and the student presses the BACK key (or types the word "BACK") he is branched immediately to the unit named in this command.

NOTE: On earlier HYPERTUTOR and TUTOR-IV systems, BACK entered a *review* mode, from which the student exited by pressing the shifted BACK key. This no longer happens. This is consistent with PLATO-IV.

BASE

SAMPLE: BASE
BASE N2,X,X,HELLO,X

TAG: Blank, a unit name, or a conditional list of unit names.

EFFECT: If blank -- specifies that the current unit is the *base* unit; no return (from a HELP type sequence) will be made in this case. All JOIN and DOs in effect are cleared.
If a unit name or conditional list -- the named unit is made the base unit. Any return will be made to that unit.

BLOCK

SAMPLE: BLOCK N1,N50,10

TAG: Three parts -- "from" variable, "to" variable, length expression.

EFFECT: This command moves HYPERTUTOR variables. Moving begins by picking up the "from" variable and moving it to the "to" variable (destroying what was previously there, but not changing the "from" variable). Then the next higher (N2 in the case of the sample) variable is moved (to N51). This continues until the number of variables specified by the last expression has been moved. Variables may be moved to and from common-blocks as well. Moving a variable which doesn't exist (N0, for instance, N151, etc.) will be ignored by HYPERTUTOR. *It will cause an execution error on PLATO-IV.*

***
BOUNDS

SAMPLE: BOUNDS 100,100
BOUNDS -100,-100,100,100

TAG: Same as for AXES.

EFFECT: This command sets limits for subsequent graphing but does not draw axes on the screen. Works only on PLATO-IV terminal.

BUMP

SAMPLE: +-x/

TAG: Characters.

EFFECT: The characters given will be deleted from the student's response before answer-judging begins.

NOTE: If one wishes to bump blanks, the blank must appear either before the other characters or between two of them. If one wishes to bump only blanks, it is necessary to bump some other harmless character as well (such as period). This command follows the associated ARROW and preceeds all answer-judging commands.

CALC

SAMPLE: CALC N5=77+N78/4
CALC N1=N2=N3=N4=5

TAG: An expression with assignment of its value to some HYPERTUTOR variable indicated. Assignment is indicated by an equal-sign (=). *PLATO-IV uses an assignment arrow (←).*

***

EFFECT: The expression to the right of the assignment symbol is first evaluated. The order of precedence is 1) expressions within parentheses are first evaluated, then 2) all functions and subscripts, 3) exponentiation (signalled by the "sup" key on PLATO-IV terminals or by an upward arrow on others), 4) multiplications before 5) divisions, 6) all additions and subtractions in left to right order, and 7) finally any "logical" operations or comparisons (EQ,NE, LT,GT,LE,GE, 8) operators AND,OR, and 9) the assignment itself (=). Character-constants may be enclosed in single quotes (to produce a left-justified constant), or in double quotes (to produce a right-justified constant).
Functions -- Some functions are specified as binary (in a mathematical sense, because they have two arguments) functions; those are CLS,ARS,$UNION$ and $MASK$, meaning "circular left shift", "arithmetic right shift", "logical union" and "masking". The logical operator NOT is a unary function which applies to the expression which follows it. Regular arithmetic and trigonometric functions implemented currently are SIN, COS, LOG, LN (natural log), ABS (absolute value), EXP, INT (integer part or truncation), and SQRT (square root), (not available on PLATO-IV).

***

Special variable names --

***

ANSCNT = number of answer-judging commands executed before matching last student answer.

***

TERMTYP= type of terminal the student is using. (see section on "wierdo terminals")

***

LESSON = name of current lesson (not PLATO-IV)

***

JCOUNT = length (in characters) of last answer

***

COST = running cost of job (not PLATO-IV)

***

TIMEON = BCD time student signed into HYPERTUTOR

***

ERROR = error code from last operation or CALC

***

NAME = student's name (20 characters)

Much more is said about CALC in the section of this booklet dealing with the TUTOR-IV language.

CALCC

SAMPLE: CALCC N5,N7=0,N7=1

TAG: Expression, followed by a *conditional list*.

EFFECT: The expression is evaluated. If its value is negative, the first assignment in the conditional list is performed. If it is zero, the second is performed, one, the third, etc. The last entry in the list will be performed for values greater than those specified by the list. In the case of the sample above, N7 will be set to zero it N5 is negative, and N7 will be set to one if N5 is greater than or equal to zero.

NOTE: Sometimes an author wishes *nothing* to take place for specific values in a CALCC. This can be easily provided for by putting the entry "N1=N1" into the appropriate place in the conditional list.

CHAR

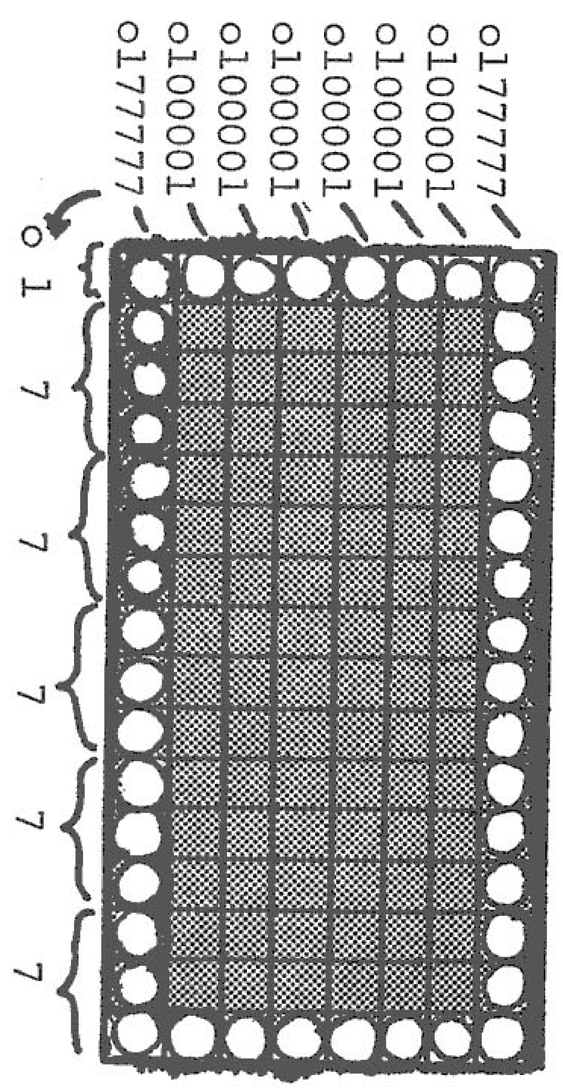
SAMPLE: CHAR 16
o177777,o100001,o100001,o100001
o100001,o100001,o100001,0177777

TAG: A number or expression telling which of the 125 available character positions in the student terminal is to be loaded with this character, followed by eight octal numbers, created as indicated below.

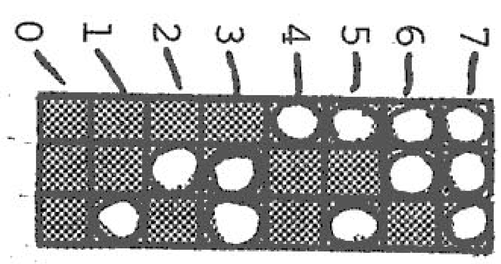
EFFECT: The eight octal specifications are loaded into the student's terminal memory as the indicated character. The CHAR command must be executed before the character can be plotted (by a PLOT command) on the screen.

NOTE: Each octal specification is preceeded by an "oh" -- see the sample above. The sample will plot the character shown below:

The character grid is always sixteen dots high by eight dots wide. One creates a character on graph paper or special forms first; then, taking each column of the character (from left to right), groups the dots by threes (except for the first dot, which is by itself). These groups of three are then converted into their octal representations to create the final CHAR command.



Octal representations.



CIRCLE

SAMPLE: CIRCLE 40,460,460
CIRCLE 4 ,460,460,0,90

TAG: Radius of circle (in dots), fine grid "X" (horizontal) and then "Y" (vertical) positions for center of circle. Optional fourth and fifth fields specify angular position of start and stop of arc (specified in degrees without the degree sign), measured counterclockwise from the positive X-axis.

EFFECT: This command will draw either a full circle or a circular arc.

*** WARNING: This command is being implemented -- potential users should check on its current status by calling 492-5367.

CIRCLEB

NOTE: This command is specified exactly as CIRCLE, and will draw a broken (dashed) circle.

*** WARNING: This command is being implemented -- potential users should check on its current status by calling 492-5367.

CLOCK

SAMPLE: CLOCK N77

TAG: A HYPERTUTOR variable.

EFFECT: This command gets the current clock time, in characters (09.10.15, for example, means 9 hours, 10 minutes and 15 seconds) and stores it in the HYPERTUTOR variable you name. This time may be displayed using SHOWA.

CLOSE

SAMPLE: CLOSE N25,N24

TAG: A HYPERTUTOR variable followed by a comma and an expression.

EFFECT: This command loads the student-response area with characters which may then be response-checked or otherwise manipulated. HYPERTUTOR looks at the variable in the command's tag and removes the rightmost six bits (which are assumed to be a character placed there by OPEN or by a suitable CALC) which it uses as the first response character. It then moves to successive variables (N26, N27, etc. in the sample). The second expression in the tag indicates how many characters are to be obtained in this fashion. The variables remain unchanged, the previous student answer is replaced for answer-judging purposes but is not changed on the screen.

CODEOUT

SAMPLE: CODEOUT o10

TAG: One of the following octal codes:

o10 = backspace	o14 = form feed
o11 = tab	o15 = carriage return
o12 = line feed	o16 = locking superscript
o13 = vertical tab	o17 = locking subscript

EFFECT: The "uncover" code (o77) is sent, followed by the specified special code. CERL Report X-15 describes the effect of these codes on the PLETO-IV terminal. This command works only on a PLATO-IV terminal!

COMMON

SAMPLE: COMMON JOE,5000 $$ *(corresponds to PLATO-IV)*

*** COMMON N1,n2*6 $$ *(not available on PLATO-IV)*

TAG: Common block name (HYPERTUTOR sill also accept an expression including a HYPERTUTOR variable), comma, length of block to be loaded (HYPERTUTOR will accept an expression, PLATO-IV will not).

EFFECT: First type -- as on PLATO-IV, a common-block, which may be accessed by several lessons or several student copies of a lesson, is initialized when the lesson is loaded. The length of the block is permanent and must remain fixed. A block is initialized with zeros when created, and any changes are "permanent" in that they are preserved after students leave the lesson.

NOTE: The common-block is the structure used for permanent storage of data by HYPERTUTOR (contrast this with the "file" concept used by other languages). A common-block is accessed as if it were simply more variables available to the program, and acts as a limited kind of "virtual storage". The users need not worry about where it is stored.

*** NOTE: Second tag type -- HYPERTUTOR allows blocks to be specified and loaded during execution of a lesson. Because only one block may be used at a time, execution of a COMMON command will first return any block in use, then load the newly specified block. This type of COMMON command must not be used by programs to be used on PLATO-IV.

COMPUTE

SAMPLE: COMPUTE V5,N10,9

TAG: A HYPERTUTOR variable, followed by another variable and an expression.

EFFECT: HYPERTUTOR will access characters stored in the second variable named and interpret them as a CALC-type expression to be evaluated. The last expression tells how many characters long the CALC-expression is. After evaluation, the result is stored in the first variable named in the tag. If the expression is invalid, a result of zero is returned and the ERROR variable is set.

NOTE: This command is usually used to evaluate expressions typed by the student and stored using STOREA.

CONCEPT

SAMPLE: CONCEPT GOES HOME
CONCEPT POISON

TAG: A series of words desired as a correct response.

PERFORMED: Only following an ARROW command. The correct structure is illustrated earlier in this booklet. A number of ANSWER, WRONG or CONCEPT commands usually follow a single ARROW. They may appear in a JOINed unit.

EFFECT: This command is used in conjunction with VOCAB, and both are explained elsewhere in this booklet. If no VOCAB is specified, CONCEPT performs like ANSWER, without the "keyword" ability. CONCEPT will mark student answers "ok" if they match.

DATA
DATA1

SAMPLE: DATA AID4
DATA1 N5,XX,YY,AID5

TAG: A unit name or conditional list of unit names.

EFFECT: When the student presses the DATA key (DATA1 means "shifted DATA" key) a branch will be made to the indicated unit. Control then proceeds through units until an END command is reached, at which point control returns to the unit in which DATA was pressed (the BASE unit).

NOTE: This command must appear in every unit in which the DATA key is to be activated. DATA, LAB and HELP function in much the same way.

DATE

SAMPLE: DATE N77

TAG: A HYPERTUTOR variable.

EFFECT: The date (in characters) is put into the indicated variable. Compare this with CLOCK, which gets the current time.

DEFINE

SAMPLE: DEFINE X=V1,Y=V2,JOE=7

TAG: A series of "definitions". A definition is a previously undefined name of one to seven characters (like "JOE" above) followed by an equal sign and either a single number, a HYPERTUTOR variable name or an expression.

EFFECT: Defined names may be used in CALCs and other expressions throughout a lesson and will be replaced by their definitions wherever used. A definition may contain *previously defined* names, if desired.

DELTA

SAMPLE: DELTA .5

TAG: An expression.

EFFECT: Sets increment of the independent variable for following FUNCT commands.

*** WARNING: This command is being implemented -- call 492-5367 to check its current status.

DO

SAMPLE: DO REPEAT,V4=3.1,5.9,.01
DO COUNT,N1=1,10

TAG: A unit name, starting assignment, ending value and increment (which is optional, and assumed to be 1 is unspecified).

EFFECT: "DO" repeatedly JOINs a set of commands, incrementing an "index" variable and testing its value against a limit. The increment may be negative, and need not be an integer. Because DO uses the JOIN command to execute the specified unit, DO and JOIN combinations may not be nested more than 10 deep.

*** WARNING: Do not use common-block variables in any DO command. This is a temporary restriction.

NOTE: Authors using DO or JOIN for plotting or calculations should know about the LOOP command.

$$ [dollars]

SAMPLE: CALC N1=1 $$ THIS IS A COMMENT!

EFFECT: Double-dollars in a line terminate the tag field of any command. The rest of the line is treated as a comment.

77

DOT

SAMPLE: DOT 1005
DOT 100,200

TAG: Coarse or fine coordinates, as for AT.

EFFECT: This command puts a single dot at the lower left hand corner of the character plotting grid (or at the fine) position specified in the tag.

DRAW
DRAWF

SAMPLE: DRAW 101,164,3264,3201,101
DRAWF 0,0,511,511

TAG: For DRAW, a series of coarse coordinates. May also include the word "skip". For DRAWF, a series of "X" and "Y" coordinates, for fine coordinates.

EFFECT: A line is drawn from the first position specified to the second, from there to the third, etc., until no more specifications remain. When the word "skip" is encountered, the screen position is changed without drawing a line.

*** NOTE: *The PLATO-IV DRAW command uses the following conventions: A DRAW in which coordinates are separated by semi-colons is interpreted in coarse coordinates. A DRAW in which coordinates are separated by commas is interpreted in fine coordinates. There is no DRAWF on PLATO-IV. It is anticipated that conversions such as these will be done by a conversion program when it becomes necessary to transfer HYPERTUTOR programs to PLATO-IV systems.*

END

SAMPLE: END
END LESSON

TAG: Blank or the word "lesson".

EFFECT: End with no tag indicates that when the student next presses NEXT (or the appropriate carriage-return key) he is to be returned to the last used BASE unit. This is used to end a HELP, DATA or LAB sequence.
END LESSON will terminate the lesson when encountered. It may be answer-contingent.

ENDARROW

SAMPLE: ENDARROW

EFFECT: This command is used to limit the range of an ARROW. It may follow the last answer-judging command for an ARROW if another ARROW will follow in the same unit. An example:

```
{ARROW
 ANSWER
 ANSWER
 WRONG
 ENDARROW
{WRITE      Here's your second question...
 ARROW
```

In the example above, were it not for the ENDARROW command, the WRITE which follows it would only be done if the student gave a wrong answer -- because of the ENDARROW, the WRITE goes with the next ARROW rather than with the WRONG.

ENTRY

SAMPLE: ENTRY JOE

TAG: A name (beginning with a character) consisting of from one to seven characters and numbers.

EFFECT: Declares an alternate entry point in a unit, so that unit may be "begun" elsewhere than at its beginning. When commands are being executed in a unit and an ENTRY command is reached, nothing happens -- however, the unit may be JUMPed to from outside and entered at that point as if it were a UNIT command.

NOTE: ENTRY names, like UNIT names, must be unique and must not duplicate UNIT names.

ERASE

SAMPLE: ERASE
ERASE 40

TAG: Blank or an expression.

EFFECT: Blank tag -- the screen is immediately erased when this command is encountered.
Expression -- the indicated number of characters is erased (covered by writing blanks), beginning at the current screen position. This type of ERASE usually follows an AT command.

NOTE: A negative or zero tag value causes the command to be ignored.

EXACT

SAMPLE: EXACT love

TAG: A word, phrase or series of characters, excluding "(" or ")".

EFFECT: A response-judging command, like ANSWER, which indicates an *exact* answer to be given to a question. The student's answer must exactly match the tag of this command for an "ok" judgment to be given.

EXIT

SAMPLE: EXIT 2

TAG: Blank or an expression.

EFFECT: Blank tag -- terminates all JOINs and DOs in effect and exits to the highest level where processing continues with the next command.
Expression -- terminates the indicated number of JOINs or DOs.

NOTE: An EXIT encountered when no JOIN or DO is in effect is ignored.

FIND

SAMPLE: FIND 0,N1,25,N26

TAG: Expression giving a value to be searched for, followed by a HYPERTUTOR variable, an expression giving the length of the proposed search, another HYPERTUTOR variable, and (optionally) an expression yielding a mask value.

EFFECT: HYPERTUTOR first evaluates the first expression and saves it. Then, searching begins at the variable specified (which may be in a common block. If the value of one of the variables is the same as the first expression given, searching terminates and a "result" is stored in the variable which follows. (N26 in the sample.) This "result" will be equal to zero if the match was found in the first comparison (with N1 above), 1 if the match was with the second (with N2 above), etc. If no

(FIND continued)

match is found, the variable is set to -1. When a mask expression is given, that mask will be "and-ed" ($MASK$) with the expression FIND is searching for and the value of the variable before a comparison is made.

FINISH

SAMPLE: FINISH LAST
FINISH N5,X,X,LAST,LASTX,X

TAG: A unit name or conditional list of unit names.

EFFECT: This command specifies a unit to be performed when the student terminates the lesson prematurely. The unit cannot contain any display commands ARROWS or PAUSEs.

NOTE: The purpose of this command is to let the lesson regain control in order to "fix up" any common block variables or records before the student ends.

*** WARNING: This command is being implemented -- call 492-5367 to check its current status.

FRAME

SAMPLE: FRAME -100,-200,100,100

TAG: Blank or four expressions giving upper limits on "X" and "Y" axes.

EFFECT: A frame will be drawn around boundaries set by previous AXES or BOUNDS commands. The four-expression tag causes the frame to be drawn around the limits specified.

FROM

SAMPLE: FROM N1;less0,unit0;less1,unit1;less2,unit2

TAG: A result variable and a conditional list of lesson names.

EFFECT: FROM first obtains the name of the lesson which JUMPOUTed to the current lesson. It then attempts a match with the conditional list. If none is found, -1 is set into the variable given ("N1" in the example). If the first name in the list matches, 0 is returned, if the second, 1 is returned, etc. Unit names may be omitted.

FUNCT

SAMPLE: FUNCT X*X,X
FUNCT X*X+2*X+4,X

TAG: An expression in terms of an independent variable, followed by a comma and the name of the independent variable.

EFFECT: This command plots an entire graph of a function as quickly as possible. The increment is taken from the last DELTA command. Existing graph bounds will be used. The independent variable is always plotted in the horizontal direction (called "X").

*** WARNING: This command is in the throes of implementation -- call 492-5367 to check its current status.

GOTO

SAMPLE: GOTO RUSE

TAG: A unit name or conditional list of unit names.

EFFECT: Processing will be continued in the unit named in the tag of this command. Unlike JUMP, neither the base or main unit pointers will be changed. When the end of the GOTOed unit is reached, the next unit to be done is the one following the unit where the GOTO was encountered. Unlike JUMP, the screen is *not* erased.

HELP
HELP1

SAMPLE: HELP AID4
TAG: A unit name or conditional list of unit names.
EFFECT: When the student presses the HELP key (HELP1 means "shifted HELP" key) a branch will be made to the indicated unit. Control then proceeds through units until an END command is reached, at which point control returns to the unit in which HELP was pressed (the BASE unit).
NOTE: This command must appear in every unit in which the HELP key is to be activated. HELP, DATA and LAB function in much the same way.

*** NOTE: *(The PLATO-IV terminal has special keys marked HELP, DATA and LAB. Most other terminals do not. Therefore, a simulated HELP key is provided. If the student types exactly "HELP" as his answer to an ARROW, this will be interpreted as a press of the HELP key. Similarly, DATA, LAB and STOP are simulated by typing their names. If answer-judging commands are present which check for any of these special names, however, the simulated keys are deactivated for that particular ARROW.)*

INHIBIT

SAMPLE: INHIBIT ARROW
INHIBIT ERASE
INHIBIT
TAG: Blank or one of the words "arrow" or "erase".
EFFECT: Blank tag -- all inhibitions are removed (sounds like fun). When the arrow is inhibited no arrow will be plotted on the screen. This makes it possible to design your own "arrow" to be plotted, using a combination of AT and WRITE or PLOT. When erasing is inhibited, the next full-screen erase will be suppressed. This makes it possible to inhibit the automatic erase done by a JUMP (simply place INHIBIT ERASE just before the JUMP command).

JOIN

SAMPLE: JOIN MORSE
TAG: A unit name or conditional list of unit names.
EFFECT: Commands in the named unit are performed "in place of" the JOIN, and then processing continues with the command following the JOIN. This command is very powerful because it makes it possible to "use" a unit from many places in a lesson simply by inserting JOIN commands where you want to "use" the unit.

*** NOTE: A form of the JOIN command with *parameters* or *arguments* is being implemented, and exists on PLATO-IV.

JUDGE

SAMPLE: JUDGE NO
TAG: A judgement ("ok","no" or "ignore") or a conditional list of judgements (possibly including "x", which means "no judgement").
EFFECT: The current judgement of the student's response is changed to the one mentioned. "ignore" causes the last response to be erased and the arrow repeated.

JUMP

SAMPLE: JUMP AWAY
JUMP N56,X,X,JOE,SAM,PETE,X

TAG: A unit name or conditional list of unit names.

EFFECT: JUMP causes an immediate change of base unit -- the unit named in its tag immediately begins executing. The screen will be erased first, unless an INHIBIT ERASE command has been executed. JUMP clears any JOINs which may still be in effect. (A JUMP in a JOINed unit, therefore, aborts the automatic return which would have been made to the command following the JOIN.)

NOTE: If the unit named in the tag of a JUMP does not exist, an execution error will be noted and the lesson will stop.

NOTE: A JUMP in a HELP-type sequence does not reset the BASE unit. The proper return will be made as soon as an END is reached.

JUMPOUT

SAMPLE: JUMPOUT "LESSON2"

TAG: A HYPERTUTOR expression giving a lesson name, and (optionally, a second expression giving a unit name within that lesson).

EFFECT: HYPERTUTOR immediately switches lessons, jumping to the new one without initializing student variables. The FROM command will show the first lesson to be the lesson of origin. Common blocks to be used in the second lesson will replace any block loaded in the first lesson.

LAB
LAB1

SAMPLE: LAB AIDS
LAB N5,AID0,AID0,AID1,AID2,X

TAG: A unit name or conditional list of unit names.

EFFECT: When the student presses the LAB key (LAB1 means "shifted LEB" key) a branch will be made to the indicated unit. Control then proceeds through units until an END command is reached, at which point control returns to the unit in which LAB was pressed (the BASE unit).

NOTE: This command must appear in every unit in which the LAB key is to be activated. LAB, HELP and DATA function in much the same way.

*** NOTE: *See the second NOTE under HELP for information on how the LAB key is simulated in HYPERTUTOR.*

LESSIN

SAMPLE: LESSIN "math"

TAG: An expression giving a lesson name.

EFFECT: The system variable "error" will be set to -1 if the named lesson is not available on the computer at the current time. It will be set to Ø if the lesson is available.

*** WARNING: This command is in the throes of implementation -- call 492-5367 to check its current status.

LEVEL

SAMPLE: LEVEL THIRD GRADE

TAG: Information on the grade-level of this lesson, to be recorded in the HYPERTEXT.

EFFECT: Information is stored for recall by students and other authors.

*** WARNING: *This command is not available on PLATO-IV.* See the description of the AUTHOR command for further notes.

LINK

SAMPLE:
```
LINK      o2000,ATTACH(RIQS,RIQS,MR=1)
LINK      o50000,RIQS.
```

TAG: A "field-length" (usually an octal constant), comma, a control card to be executed.

EFFECT: Using "ESP", this command processes the indicated system control card in the indicated field-length. The system variable called ERROR will be set upon return if the control card caused an execution error. Possible values of ERROR are:

1.0= time limit	8.0=(rerun)
2.0= arithmetic error	9.0= control-card error
3.0= pp abort	10.0= ecs parity error
4.0= cpu abort	11.0=(job-card error)
5.0= pp call error	12.0=(pre-abort)
6.0= operator drop	13.0= auto-recall error
7.0=(kill)	14.0= job hung in auto-recall
	15.0= mass storage limit

NOTE: Error codes enclosed in parentheses above will normally not be encountered by online jobs.

*** WARNING: *This command is not available on PLATO-IV.*

MODE

SAMPLE:
```
MODE      ERASE
MODE      REWRITE
```

TAG: One word, "erase", "rewrite" or "write".

EFFECT: On the PLATO-IV terminal, this command specifies a change of terminal writing mode. It affects subsequent display commands. Mode *rewrite*, which is the standard mode, erases the background (at a character position), then plots the next character. Mode *write* plots without erasing the background. Mode *erase* turns dots off in the shape of the character or line selected. Upon initiation of a new main unit (via JUMP or NEXT) the mode is set to rewrite.

*** WARNING: *Mode rewrite is standard on HYPERTUTOR -- mode write is the standard on PLATO-IV.*

MODPERM

SAMPLE: MODPERM

EFFECT: This command restores the list of SETPERM numbers minus any values removed by REMOVE commands. It is used in the manipulation of random permutations.

NAME
(HYPER-TUTOR version)

SAMPLE: NAME MATH

TAG: Lesson name. May be up to 10 characters long.

EFFECT: This command declares a lesson name to be added to the HYPERTEXT. All lessons are PRIVATE (name not listed in the HYPERTEXT) unless accompanied by a PUBLIC command.

*** NOTE: *Not a PLATO-IV command! See next page for PLATO-IV version. This version must appear before the first UNIT command!*
This must be the first command of every HYPERTUTOR lesson!

NAME (PLATO-IV version)

SAMPLE: NAME N99
TAG: A HYPERTUTOR variable.
EFFECT: The sign-on name of the student is stored in the tag variable, and ending in the following variable (two are used). The name will have zero-fill.

*** WARNING: *Any NAME command appearing after the first UNIT command in a HYPERTUTOR lesson will be assumed to be this version. This is the standard version for PLATO-IV.*

NEXT

SAMPLE: NEXT ABC
NEXT N55-1,X,X,ABC,X
TAG: A unit name or conditional list of unit names.
EFFECT: This command specifies the unit which will follow in sequence when all arrows in the present unit are satisfied and the user presses NEXT. Any unit reached by a NEXT command will become a main or base unit. A NEXT in a help-type sequence does not change the BASE unit. NEXT X leaves the pointer unchanged. *The message "PRESS NEXT" (or appropriate keyname for non-PLATO terminals), will appear at the bottom of the screen automatically.*

NEXTNOW

SAMPLE: NEXTNOW ABC
NEXTNOW N55-1,X,X,ABC,X
TAG: A unit name or conditional list of unit names.
EFFECT: This command acts like a NEXT, except that it allows the student to continue even though all ARROWs are not judged "ok". All JOINs and DOs are cleared by NEXTNOW.

NO

SAMPLE: NO
EFFECT: This command is a judging-command. It signals that if the last student response is not yet judged either "ok" or "no" it is to be immediately judged "no". The ARROW will be repeated.
NOTE: If two "NO" commands appear following an ARROW, the first will catch all blank responses (pressing of NEXT) while the second will catch all other responses. The NO command is sometimes called a *universal NO*.

NOWORD

SAMPLE: NOWORD wrong!
TAG: Up to seven characters, letters, punctuation.
EFFECT: Sets the legend to be plotted on the screen next to a wrong student answer (in place of "no"). It will be used in all places where "no" would have been used. This only need be set once per lesson, and will be reset to "no" for any subsequent lessons.

OK

SAMPLE: OK
EFFECT: This command is a judging-command. It signals that if the last student response is not yet judged either "ok" or "no" it is to be immediately judged "ok". See the note under the NO command for the effect of two OK commands following an ARROW.

OKWORD

SAMPLE: OKWORD right!
TAG: Up to seven characters, letters, punctuation.
EFFECT: Sets the legend to be plotted on the screen next to a right student answer (in place of "ok"). See NOWORD command also.

OPEN

SAMPLE: OPEN N25,N24
TAG: A HYPERTUTOR variable followed by a comma and an expression.
EFFECT: This command is the opposite of CLOSE. It spreads a student response to allow character-by-character examination from HYPERTUTOR variable locations. The characters of the response are inserted one per variable into the starting location and subsequent variables, continuing through the number of locations given by the second expression. Characters are stored as integers (A=1,B=2,C=3,D=4, etc.). If the student typed "HELLO", for example, N25 would equal 8, N26 would equal 5, N27 would equal 12, etc., in the sample above. One can always use "JCOUNT" as the second expression to force the storing of the full student answer.

OR

SAMPLE: ANSWER yes
OR
EXACT positive
TAG: None.
EFFECT: This command really serves as a "link" between two answer-judging commands, usually of different types (such as the ANSWER and EXACT commands in the sample above). Answer-contingent commands which follow the last answer-judging command will be executed for *either* of the answer-checks linked by an OR command. Several response-judging commands may be so linked using a series of OR commands.

ORIGIN

SAMPLE: ORIGIN V1,V2
TAG: Coarse or fine coordinates.
EFFECT: This command sets the origin for subsequent graphing commands (FUNCT, AXES, BOUNDS, etc.). The origin may be changed by subsequent ORIGIN commands.

PACK

SAMPLE: PACK N2,N47,This is a sentence.
PACK N29,So is this.
TAG: Optional character-count variable , a HYPERTUTOR variable and a string of characters.
EFFECT: The length of the character string is first determined and stored into the first variable, if present. Then, the string itself is moved (ten characters at a time) into the variable named and subsequent locations as necessary.

PAUSE

SAMPLE: PAUSE 15
TAG: Blank or a HYPERTUTOR expression.
EFFECT: Blank tag -- the computer pauses until NEXT is pressed. Expression -- the computer pauses until the specified number of seconds has elapsed. This is usually accurate to about the nearset 2 seconds, but is not guaranteed.
(more on the next page)

(PAUSE continued)

The PAUSE command is timed from the point at which it is executed, which may be before the terminal screen has been completely written. Therefore, the time specified should normally include any time necessary to complete screen displaying. HYPERTUTOR accomplishes the PAUSE by asking the operating system to rolloff the job until the specified time has passed -- rollon will occur as soon as possible after that time, but may take a few more seconds.

PLOT

SAMPLE: PLOT

TAG: A number or expression telling which of the 125 available character positions in the student terminal is to be plotted at the current screen position. (Usually these will have been loaded by previously processed CHAR commands.)

EFFECT: Dots specified by a previous CHAR command are lit (or erased if the terminal is in mode *erase*). Screen position is moved one character to the right, so successive PLOT commands will appear left-to-right as if written by a WRITE command.

PRIVATE

SAMPLE: PRIVATE

EFFECT: This HYPERCOMMAND (which must follow NAME and preceed the first UNIT of a lesson) specifies that a lesson is not to appear in the list of PUBLIC lessons which students will see.

*** NOTE: All lessons are declared PRIVATE unless otherwise specified, so this command is seldom used. *Not used on PLATO-IV.*

PUBLIC

SAMPLE: PUBLIC

EFFECT: This HYPERCOMMAND (which must follow NAME and preceed the first UNIT of a lesson) specifies that a lesson is ready for student use and that the author desires to have it listed in the completed HYPERTEXT. *Not used on PLATO-IV.*

*** NOTE: Currently, any lesson declared PUBLIC by its author is listed for students. At some future time, lessons will undergo a review process and will be declared fully public only after a "provisional" period, during which they can be tested by students. This does not imply that the lessons cannot be used until they are PUBLIC -- *all* lessons, public or private, are always available to the student who knows their names.

RANDP

SAMPLE: RANDP N44 $$ "P" stands for "permutation"

TAG: A HYPERTUTOR variable.

EFFECT: This command randomly selects an integer from the permutation list generated by the last SETPERM command. This corresponds to sampling without replacement. When a set of SETPERM numbers is exhausted, the value zero will be returned by RANDP.

NOTE: Execution of another SETPERM command always establishes a new selection pool (permutation).

RANDU

SAMPLE: RANDU V34
RANDU N34,10

TAG: First type -- a HYPERTUTOR "V" type variable.
Second type -- a HYPERTUTOR "N" type variable and an expression.

(more on the next page)

REMOVE

SAMPLE: REMOVE N8-99
TAG: An expression.
EFFECT: The number corresponding to the value of the expression is removed from the SETPERM-created permutation and from future sets of MODPERM numbers. (A new SETPERM will clear this condition.)

SETPERM

SAMPLE: SETPERM 52
TAG: An expression.
EFFECT: A list of numbers in "random" order is set up. The numbers will range from 1 to the tag value. The sample would set up all numbers from 1 to 52 (included) in random order, available for use by RANDP commands. The first RANDP command will get the first available number -- subsequent RANDPs will get more numbers between 1 and 52 -- a single number is never repeated. When the list is exhausted, further RANDP commands will retrieve zeros.

SHOW

SAMPLE:
```
SHOW      N5,6
SHOW      V62,6.2
SHOW      N(N77+1),N55
```
TAG: An expression or variable whose value is to be shown on the screen, followed by an expression giving the "format".
EFFECT: The expression or variable is evaluated and shown in the specified format *beginning at the current screen position.* Formats -- for "N" type variables or expressions, the format is a single integer number indicating how many spaces (maximum) the number is to be allotted on the screen. If the number will not fit, asterisks will be printed instead. In the first example above, N5 will be printed in 6 spaces. If 6 spaces are not required, the number will be right-justified.

-- for "V" type variables or expressions, the format will be a number with a single digit after the decimal point. The "whole number" part of this indicates how many spaces are to be allotted before the decimal point in the shown number, and the "fractional" part indicates how many spaces are to be allotted after the decimal point. In the second sample above, 6 spaces before the decimal and 2 after it make a total of 9 spaces for the number (including 1 for the decimal point itself).
NOTE: A SHOW always continues from the current screen position and normally will follow an AT command. Successive SHOW commands will therefore appear on the same line unless specifically directed by interposed AT commands.

SHOWA

SAMPLE:
```
SHOWA     NAME,20
SHOWA     N55,8
SHOWA     'HELLO',5
```
TAG: An expression whose value (in characters) is to be shown, and a second expression telling how many characters are to be processed.
(more on the next page)

(SHOWA continued)

If the number of characters specified for processing is greater than ten, processing will continue in successive HYPERTUTOR variables. If the item to be shown is a true expression (not a variable), a maximum of ten characters may be shown. If a character length is not specified, it is assumed to be ten. Any "hard" zeros (binary zero) found in the material being processed will be ignored. [For example, if a variable to be shown contains the number one, in binary, it is actually interpreted as a character "A", right-justified in the word. When shown, all of the binary zeros which preceed the "A" will be skipped, although they are counted, and only the single "A" will be shown, as if it were left-justified on the screen.]

SHOWE

SAMPLE: SHOWE V1,5

TAG: An expression or variable whose value is to be shown on the screen, followed by an expression giving the "format".

EFFECT: The expression or variable is evaluated and shown as for SHOW except that an exponential notation (scientific notation) is used. The "format" part indicates how many digits are to be shown after the decimal point. A showe of the value 184.0, using the "5" format of the sample would show:

1.84000×10^2

*** NOTE: Exponentiation is indicated by superscripts on the PLATO-IV terminal and by an uparrow (↑) on others.

SHOWO

SAMPLE: SHOWO N7,6

TAG: An expression and a "format" expression.

EFFECT: The value of the expression or variable is shown as by SHOW except that conversion is performed to the base 8 rather than 10. This is known as octal conversion. Leading zeros are shown, in contrast to SHOW, which suppresses leading zeros.

SLIDE

SAMPLE: SLIDE 4

TAG: An expression.

EFFECT: The indicated slide (numbered from 1 to 256) is selected and shown. A number greater than 256 will turn the projector off. The projector is also turned off at the end of a lesson.

SPECS

SAMPLE: SPECS NOOKNO

TAG: Blank or "nookno".

EFFECT: If the tag is "nookno", the "ok" or "no" judgements are not printed when the student answers a question. The effect of this command lasts through the current ARROW only.

STATUS

SAMPLE: STATUS COMPLETE

TAG: Anything indicating the completion status of the lesson.

EFFECT: This HYPERCOMMAND (which must follow NAME and preceed the first UNIT of a lesson) specifies the status of the lesson and is available to students browsing through HYPERTEXT.

*** WARNING: *This command is not available on PLATO-IV.*

STORE

SAMPLE: STORE N77
TAG: A HYPERTUTOR variable.
EFFECT: The student's response is treated as an expression to be evaluated. The result of evaluation is placed in the variable named in the tag of this command. If the expression is improperly constructed, it is judged "no" and the arrow is repeated.
PERFORMED: After an ARROW and before the first response-judging command.
NOTE: If you use STORE, the student must type a number or expression and nothing else, in his answer. Also see note under STOREN.

STOREA

SAMPLE: STOREA N77,10
TAG: A HYPERTUTOR variable and an expression.
EFFECT: This command stores the student's response as a character string, in the variable named. The number of characters stored is determined by the value of the second expression. "JCOUNT" is often used as the length, in which case all of the characters in the student's response will be stored (no more, no less).

STOREN

SAMPLE: STOREN N77
TAG: A HYPERTUTOR variable.
EFFECT: As for STORE, a number or expression is converted from the student's answer, *but* the characters thus converted are then removed from the answer, so that several consecutive STOREN commands will remove consecutive numbers from the response. This is the only way the student can type more than one number per line in his answer and have it correctly interpreted.
NOTE: STORE and STOREN will be treated as answer-judging commands if the expression is improper -- that is, commands which follow them will be executed if the answer is judged "no". This makes it convenient to respond to incorrect expressions by placing WRITE commands after a STORE or STOREN.

*** WARNING: This command is being implemented -- call 492 5367 to check its current status.

SUBJECT

SAMPLE: SUBJECT ELEMENTARY MATHEMATICS,
INTRODUCTION TO SETS.
TAG: Multiple-lines, giving a short (less than 5 lines) description of the subject of this lesson.
EFFECT: This HYPERCOMMAND must follow the NAME command and preceed the first UNIT. It gives information to be entered into the HYPERTEXT.

*** WARNING: *This command is not used on PLATO-IV.*

SUB1

SAMPLE: SUB1 N70
TAG: A HYPERTUTOR variable.
EFFECT: This command subtracts 1 from the value of the tag variable.

TERM

SAMPLE: TERM WEIGHT

TAG: A character string, single word, whose first seven characters are used as a marker.

EFFECT: All TERM commands occurring within a lesson are gathered together into a directory. Each TERM is associated with the UNIT in which it resides. When a student presses the key marked "term" (or types "term" on non-PLATO-IV terminals) he is asked which term he wishes to access (there is no directory, so the student must be informed about possible options). If the student gives a response in which the first seven characters match the tag of one of the TERMs, the lesson will immediately branch to the unit associated with that TERM.

NOTE: TERM can be used as a dictionary of definitions, a special author-aid or many other things.

*** NOTE: *PLATO-IV accepts the first 8 characters as a TERM name and does not allow multiple word TERMs.*

UNIT

SAMPLE: UNIT ABC

TAG: A unit name -- one to seven alphabetic and numeric characters. Each unit name must be unique within a lesson. The names "X" and "Q" may not be used.

EFFECT: Declares the start of a section of the lesson. Branching commands, such as JUMP, GOTO, NEXT, all refer to units which have been defined by UNIT commands. A UNIT is the smallest part of a lesson which the computer can actively jump to -- once in a UNIT, the computer processes commands in order or as specified by the interaction between ARROW and its associated answer-judging commands.

VOCAB

SAMPLE: VOCAB (yes,certainly,positive,sure,will,would)
(no,don't,can't,won't,not,negative)⟨a,an,the⟩

TAG: See special section on the VOCAB command and how it interacts with CONCEPT during answer-judging. Lists of synonyms are enclosed within parentheses. Lists of ignorable words within angle-brackets.

EFFECT: Lists of synonymous words or phrases are used during CONCEPT answer-judging. The author specifies any word from a "ring" of synonyms and HYPERTUTOR will accept the student's replacement by any of the other synonyms within the ring.

*** WARNING: There is a limit of about 400 words per VOCAB in HYPERTUTOR.

WRITE

SAMPLE: WRITE This is a paragraph
of interesting information.

TAG: One or more lines of characters to be displayed on the screen.

EFFECT: Characters and numbers are written on the screen beginning at the current screen position. Usually WRITE is preceeded by an AT command. When the WRITE continues onto subsequent lines, it is called a *paragraphed* WRITE and the left margins of subsequent lines all line up vertically with the top line, as if each following line had its own individual AT command in front of it. (more on the next page)

(WRITE continued)

WRITE and SHOWs can be intermixed in a convenient way. If a WRITE command follows a SHOW, the two will be displayed on the same line, provided no AT commands intervene. And, if SHOW follows WRITE, the same thing happens. Thus...

```
WRITE     A
WRITE     B=
SHOW      V7,1.2
```

(if V7=4.3, for example) would show...

```
AB=4.3
```

...on the student's screen. However,

```
WRITE     A
          B=
SHOW      V7,1.2
```

...would show...

```
A
B=4.3
```

...because of the paragraphed nature of the WRITE command. Positioning can always be adjusted with AT commands.

WRITEC

SAMPLE:

```
WRITEC    N8,minus,zero,one,two,three or greater
WRITEC    N8,minus
          zero
          one,two,three or greater
```

TAG: A conditional list of things to be written.

EFFECT: The first argument is an expression or HYPERTUTOR variable, which is to be used to determine which of a number of words or sentences will be displayed. (Don't use "X" in the list because it will get printed -- just leave the list element blank.) Commas or carriage-returns can be used to separate the list elements. The first WRITEC example above is exactly equivalent to the second.

WRONG

SAMPLE: WRONG (hate,enmity)

TAG: A list of words or phrases which constitute anticipated incorrect answers to the last question asked. Parentheses are used to enclose synonymous answers, as for ANSWER.

PERFORMED: Only following an ARROW command, with other response-judging commands. See ANSWER.

EFFECT: Exactly as for ANSWER, except that matched answers will be judged "no" (or as specified by the NOWORD command).

WRONGV

SAMPLE: WRONGV N5,9

TAG: Exactly the same as for ANSV.

EFFECT: Exactly as for ANSV except that a "no" judgment will be made.

WRONGVA

SAMPLE: WRONGVA N55,12

TAG: Exactly as for ANSVA.

EFFECT: Exactly the same as ANSVA, except for a "no" judgment.

ZERO

SAMPLE: ZERO N7,3

TAG: A HYPERTUTOR variable and an optional expression telling how many variables (consecutively) are to be cleared.

EFFECT: Indicated variable(s) will be set to zero.